*I can do everything through him*
*who gives me strength.*
–Philippians 4:13 (NIV)

University of Alberta

# Applicability of Method Specialization Techniques to Java

by

Shane Allen Brewer

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta
Fall 2005

0-494-09131-2

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

*Your file* *Votre référence*
*ISBN:*
*Our file* *Notre référence*
*ISBN:*

NOTICE:
The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:
L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

---

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

To my mother Doreen,
my father Ed,
my grandmother Lois,
my grandfather Zeb,
my sister Jayla,
my brother Jordan, and
my love Woosun

# Abstract

Method specialization is an optimization used to eliminate virtual call sites and open up opportunities for other compiler optimizations. Existing method specialization techniques do not explicitly handle dynamic class-loading or are suitable for a dynamic compilation environment. This thesis examines previous method specialization techniques and illustrates the transformations with a running example. These techniques are also reviewed to determine the applicability of each method for use in a dynamic compilation environment that support dynamic class-loading (such as Java). Additionally a new method specialization framework is given that is designed for a dynamic compilation environment and handles dynamic class-loading. Aspects that need to be examined when making method specialization decisions for a dynamic compiler are listed and analyzed. Finally numbers regarding opportunities for method specialization the SPECjvm98 and SPECjbb2000 benchmarks suites are listed and investigated.

# Acknowledgements

I first must recognize my Lord and Savior Jesus Christ, whose perfect love has set me free and is the source of my joy and hope. Father I appreciate all you have done for me throughout my whole life and words cannot express the gratitude for the grace you have shown me. Thank you so much.

Secondly, I must express my admiration and appreciation for my supervisor Dr. José Nelson Amaral. Nelson, your support of my desire to pursue my swimming dreams and therefore the corresponding schedule really meant the world to me. You were there to stick with me when I wanted to quit, and were constantly a source of wisdom and patience when I needed it most. I will always have a deep admiration of your personal characteristics as well as your extensive knowledge.

I am so thankful to my parents, Ed and Doreen, who have supported me regardless of the decisions that I've made. Your encouragement and financial help have been a constant support for me and I have always felt that you have been my safety net. Without your support, there is no way I would have been able to complete this goal.

I am eternally indebted to my grandparents, Zeb and Lois. Your generosity and giving continue to amaze me and your unconditional love has set the standard for my life. My ultimate goal is to be half person you both are to me.

My brother Jordan, I am thankful that you are there to balance out my life with love, laughter and video games. Your personality and character constantly help me expand my horizons and perceptions and the memories of watching "Whose Line is it Anyway" and playing "Halo" will stay with me forever.

My sister Jayla, you are definitely taking after Zeb and Lois with your generosity and love. Your heart and caring shine through in your beautiful personality. You were always there for me, no matter what. I can honestly say that no one could have a better sister than you.

And finally to my wife to be, Woosun. You have constantly amazed me with

your selflessness, your understanding, and your love. At points I have trouble understanding how you could be so understanding with my schedule of research, swim practices, swim meets, and training camps. Every day I grow in awe of your character and grow in love for your heart. I am so lucky to have the privilege of being with you. I love you.

# Table of Contents


**1 Introduction** 1
- 1.1 Introduction . . . 1
- 1.2 Known Compiler Techniques . . . 2
- 1.3 Summary of Major Contributions . . . 2
- 1.4 Outline . . . 3

**2 Definitions** 5

**3 Overview of The Jikes Research Virtual Machine** 24
- 3.1 Booting Jikes . . . 24
- 3.2 Object and Memory Layout . . . 25
- 3.3 Memory Management Subsystem . . . 25
- 3.4 Intermediate Representation . . . 27
- 3.5 Compiler Subsystem . . . 30
  - 3.5.1 The Baseline Compiler . . . 31
  - 3.5.2 The Optimizing Compiler . . . 31
  - 3.5.3 The Adaptive Optimization System . . . 31
- 3.6 Inlining . . . 34
  - 3.6.1 Optimizing Compiler Inlining Overview . . . 34
  - 3.6.2 Adaptive Compiler Inlining Overview . . . 34

**4 An Overview Of Method Specialization** 36
- 4.1 A Introduction To Method Specialization . . . 38
- 4.2 Method Specialization And Call Site Devirtualization . . . 38
- 4.3 Method Specialization On Other Program Properties . . . 39
- 4.4 Under-specialization and Over-specialization . . . 41
- 4.5 Method Specialization Selection Strategies . . . 43
- 4.6 Advantages and Disadvantages . . . 44
- 4.7 The Focus of This Thesis . . . 45

**5 Customization** 46
- 5.1 Description . . . 46
- 5.2 Strategy . . . 46
- 5.3 An Example of Customization . . . 47
- 5.4 Results And Issues . . . 47
- 5.5 Suitability For Java . . . 51

**6 Selective Specialization** **52**
6.1 Description . . . 52
6.2 Strategy . . . 53
6.3 Example . . . 53
6.4 Cascading Specializations . . . 59
6.5 Results . . . 61
6.6 Suitability For Java . . . 61

**7 Automatic Program Specialization** **62**
7.1 Description . . . 62
7.2 Strategy . . . 64
7.3 Example . . . 64
7.4 Results . . . 66
7.5 Suitability For Java . . . 66

**8 A New Method Specialization Framework For Java** **67**
8.1 Why A New Framework? . . . 67
8.2 Method Specialization And Java Issues . . . 67
8.2.1 The Java Super Keyword . . . 67
8.2.2 Dynamic Class Loading . . . 69
8.3 Framework Description . . . 71
8.3.1 Profiling Collection . . . 71
8.3.2 Aspects to Consider in Method Specialization Decisions . . . 74
8.3.3 Optimization Transformation . . . 75
8.3.4 Method Selection . . . 76
8.3.5 Class Loader Changes . . . 78
8.4 Costs of Performing Method Specialization . . . 79
8.4.1 Profiling Information . . . 83
8.4.2 Calculation of Decision . . . 84
8.4.3 Performing the Transformation . . . 84
8.4.4 Class Loader Checks . . . 84
8.5 Benefits of Performing Method Specialization . . . 85
8.6 Summary . . . 86

**9 Empirical Study of Devirtualization Opportunities** **87**
9.1 Benchmark Description And Discussion . . . 88
9.2 Experimental Setup . . . 89
9.2.1 Jikes Research Virtual Machine Modifications . . . 89
9.2.2 Jikes Configuration . . . 90
9.3 Static Devirtualization Opportunities . . . 91
9.3.1 Static Specialization Empirical Information Collected . . . 91
9.3.2 SPECjvm98 and SPECjbb2000 Static Devirtualization Opportunities . . . 94
9.3.3 Discussion of Static Opportunity Numbers . . . 101
9.4 Summary . . . 102

**10 Related Work** **103**
10.1 Extant Analysis . . . . . 103
10.2 Class Hierarchy Analysis . . . . . 106
10.3 Profile-Guided Receiver Class Prediction . . . . . 107
10.4 Code Patching . . . . . 109
10.5 Pre-existence Analysis . . . . . 111
10.6 Thin Guards . . . . . 112

**11 Conclusions** **115**

**12 Future Work** **117**
12.1 Collection of Dynamic Execution Information for Method Specialization Opportunities . . . . . 117
12.2 Test Additional Complex Benchmarks . . . . . 117
12.3 Removal of Specialized Methods . . . . . 118
12.4 A Heuristic for Making Method Specialization Decisions . . . . . 118

**Bibliography** **119**

**A Raw Data for SPECjvm98 and SPECjbb2000 Benchmarks** **127**

**B Trademarks** **129**

# List of Figures


2.1 A Sample Pseudo Java Class Hierarchy . . . . . 6
2.2 A Sample Pseudo Java Program Demonstrating Different Parameters 6
2.3 A Control Flow Graph . . . . . 8
2.4 A Program Call Graph . . . . . 9
2.5 A Sample Pseudo Java Class Hierarchy . . . . . 10
2.6 A Virtual Method Table For The Class Hierarchy In Figure 2.5 . . . 11
2.7 A Sample Pseudo Java Class Hierarchy . . . . . 12
2.8 The Applies-To Sets For Method *foo()* For The Class Hierarchy In Figure 2.7 . . . . . 13
2.9 A Receiver-Class Distribution For the Call Site *foo()* in Method *A.func()* from Figure 2.7 . . . . . 16
2.10 Sample Java Pseudo Code Before Method Inlining . . . . . 18
2.11 Sample Java Pseudo Code After Method Inlining . . . . . 18
2.12 A Class Hierarchy Graph . . . . . 20
2.13 An Example Pointcut, After [83] . . . . . 22
2.14 An Example of Advice, After [83] . . . . . 23
2.15 An Example of an Aspect, After [83] . . . . . 23

3.1 An example TIB . . . . . 26
3.2 The Jikes RVM Intermediate Representation Transformation, after citeAlpern2000Jalapeno 28
3.3 The Jikes Adaptive Optimization System as shown in [15] . . . . . 32

4.1 A Sample Pseudo Java Class Hierarchy Before Method Specialization 37
4.2 A Graphical Representation of The Class Hierarchy From Figure 4.1 37
4.3 A Sample Pseudo Java Class Hierarchy After Method Specialization 40
4.4 A Sample Pseudo Java Method Before Data Encapsulation Method Specialization . . . . . 40
4.5 A Sample Pseudo Java Method After Data Encapsulation Method Specialization . . . . . 41
4.6 A Sample Pseudo Java Class Hierarchy Before Imperative Computation Method Specialization . . . . . 41
4.7 A Sample Pseudo Java Class Hierarchy After Imperative Computation Method Specialization . . . . . 42

5.1 A Sample Pseudo Java Class Hierarchy Before Method Specialization 47
5.2 Class *A* After Customization . . . . . 48
5.3 Class *B* After Customization . . . . . 48
5.4 Class *C* After Customization . . . . . 48

5.5 Class *D* After Customization . . . . . 49
5.6 Class *E* After Customization . . . . . 49
5.7 Class *F* After Customization . . . . . 50

6.1 A Sample Pseudo Java Class Hierarchy . . . . . 54
6.2 A Sample Weighted Call Graph For the Class Hierarchy in Figure 6.1 55
6.3 Method Reachability of method *foo()* . . . . . 56
6.4 Virtual Function Tables Before Method Specialization . . . . . 57
6.5 Virtual Function Tables After Method Specialization . . . . . 58
6.6 A Sample Pseudo Java Class Hierarchy Showing How Call Sites Can Become Virtual After Specialization . . . . . 59
6.7 A Sample Pseudo Java Class Hierarchy After Performing Cascading Method Specialization . . . . . 60

7.1 A Sample Pseudo Java Class Hierarchy . . . . . 64
7.2 A Sample Specialization Class . . . . . 64
7.3 The Corresponding Aspect Containing Specialized Code . . . . . 65

8.1 A Sample Pseudo-Java Program Using the **super** Keyword . . . . . 68
8.2 A Sample Pseudo-Java Program Using the Super Keyword After Method Specialization . . . . . 68
8.3 A Sample Pseudo-Java Program . . . . . 69
8.4 A Sample Pseudo-Java Program After Method Specialization . . . . 70
8.5 A Sample Pseudo-Java Program After Dynamic Class Loading . . . 71
8.6 A Sample Pseudo-Java Class Hierarchy . . . . . 72
8.7 Dynamically-Loaded Classes Loaded Into the Class Hierarchy in Figure 8.6 After Performing Method Specialization . . . . . 72
8.8 Sample Receiver-Type Profiling Information for Call-Site foo() . . . 73
8.9 A Sample Pseudo-Java Class Hierarchy After Method Specialization 76
8.10 The Virtual Method Table After Method Specialization . . . . . 77
8.11 The Class Hierarchy After Dynamically Loading Class *E* . . . . . 79
8.12 The Virtual Method Table After Dynamically Loading Class *E* . . . 80
8.13 The Class Hierarchy After Loading Class *D* . . . . . 81
8.14 The Virtual Method Table After Dynamically Loading Class *D* . . . 82
8.15 Methods from Class *A* in Figure 8.6 to Show Receiver Object-Type Profiling Information . . . . . 83

9.1 Number of Specializable Classes Compared to Total Loaded Classes in SPEC Java Benchmarks . . . . . 94
9.2 Number of Specializable Methods Compared to Total Loaded Methods in SPEC Java Benchmarks . . . . . 95
9.3 Number of Devirtualizable Call Sites Compared to Total Call Sites in SPEC Java Benchmarks . . . . . 96
9.4 Number of Inlinable Specializable Call Sites Compared to Total Number of Call Sites in SPEC Java Benchmarks . . . . . 97

10.1 A Sample Pseudo-Java Class Hierarchy Illustrating Extant Analysis, After [115] . . . . . 104

10.2 A Sample Pseudo-Java Class Hierarchy Illustrating Extant Analysis After Specialization, After [115] . . . . . 105
10.3 A Sample Class Hierarchy . . . . . 106
10.4 A Sample Pseudo Java Class Hierarchy from [75] . . . . . 108
10.5 A Call Site and the Corresponding Optimized Code from [75] . . . . 108
10.6 Assembly Code Before Dynamic Class Loading From [76] . . . . . 110
10.7 Assembly Code after Dynamic Class Loading that Invalidates Inlining Decision From [76] . . . . . 110
10.8 An Example Illustrating Pre-existence After [51] . . . . . 111
10.9 Sample Java Pseudo Code Demonstrating an Opportunity for Inlining, After [13] . . . . . 113
10.10Sample Java Pseudo Code Demonstrating Using Thin Guards For Inlining, After [13] . . . . . 113

# List of Tables


8.1 Example Receiver-type Profiling Information for Call Sites from Figure 8.15 . . . . . 83

9.1 Description Of Benchmarks . . . . . 88
9.2 Number of Specializable Methods per Specializable Class in SPEC Java Benchmarks . . . . . 98
9.3 Number of Devirtualizable Call Sites per Specializable Method in SPEC Java Benchmarks . . . . . 99
9.4 Number of Callees per Specializable Method in SPEC Java Benchmarks 99
9.5 Size (bytes) of Specializable Methods in SPEC Java Benchmarks . . 100
9.6 Size (bytes) of Devirtalizable Call Site Callees in SPEC Java Benchmarks . . . . . 100

A.1 Static Number of Specializable Classes . . . . . 127
A.2 Static Number of Specializable Methods . . . . . 128
A.3 Static Number of Specializable Call Sites . . . . . 128
A.4 Static Number of Inlinable Specializable Methods . . . . . 128

# Chapter 1

# Introduction

## 1.1 Introduction

Object-oriented programming languages are a popular choice for programmers as they allow easy extensibility of programs, abstraction of concepts, and reusability of code. Object-oriented languages have become more popular than in the past as compiler optimizations — inlining [51], class hierarchy analysis [50], receiver-class prediction [65], and others — have improved the execution speed resulting in code that is often comparable with their procedural language counterparts.

Method specialization is a compiler optimization used to improve object-oriented language execution speed. Multiple method specialization implementations have been researched in multiple programming languages.

This thesis reviews method specialization techniques used in object-oriented compilers and describes a new framework that can be used with object-oriented languages that support dynamic class-loading (such as Java). To the best of our knowledge, no publication has summarized and compared existing method specialization techniques, nor created a method specialization compiler optimization framework that explicitly handles dynamic class-loading.

The new framework discussed in this thesis has not been implemented yet. It is presented with the assumption that it would be implemented in the Jikes Research Virtual Machine [44]. We also use Java pseudo code for our running examples. While some the optimizations that we review were not intended for the Java language, we have chosen to display the transformation in Java for consistency. Additionally we state that the examples are pseudo code as they are not made to be compiled and run on their own, but to illustrate the optimizing transformations.

## 1.2 Known Compiler Techniques

Method specialization optimizations have been implemented in several programming languages, including: SELF [27], Sather [89], Trellis [101], C++ [116], and Java [64]. Several of these method specialization techniques have been extensively researched and published and therefore are well known within the compiler community. These techniques, which are discussed and compared here include: customization, selective specialization, and automatic program specialization.

These method specialization techniques do not explicitly handle dynamic class loading, and we are unaware of any other published material that handles this case.

## 1.3 Summary of Major Contributions

The major contributions of this thesis are:

- A summary of comparison and example of all known method specialization techniques including:
  - Customization [26]
  - Selective Specialization [48]
  - Automatic Program Specialization [109]
- A new framework for a method specialization compiler optimization framework that explicitly handles dynamic class-loading.
- Listing and discussion of aspects that need to be considered when making a method specialization decision.
- Experimental results of static opportunities for method specialization in the SPECjvm98 and SPECjbb2000 benchmark suites.
- Discussion of areas of future improvement for method specialization implementations.

The summary of previous method specialization techniques gives a summary of each know method specialization technique. Additionally, a discussion of each technique follows for the technique's applicability for modern programming languages that implement dynamic compilation and support dynamic class loading. Support is

also built to show the need for a new method specialization framework that supports these new aspects.

After showing the new for a new framework, a new method specialization compiler optimization framework is described and discussed in detail. This framework is tailored for the Java programming language, but is suited for dynamic compilation environments and explicitly handles dynamic class-loading.

Given the currently popular technique of dynamic compilation, optimization decision making becomes more and more important to maintain that the execution speedup outweighs the cost of performing the optimization. Therefore we present a discussion of the aspects that need to be considered when making a method specialization optimization decision.

We have also collected static experimental data regarding method specialization opportunities in the SPECjvm98 and SPECjbb2000 benchmark suites. We collected information on multiple areas that are essential for maximizing execution speedup when performing method specialization.

As a result of our research into all different method specialization techniques, we have a unique understanding of the areas of improvement that are needed. These areas are discussed for the researcher looking for areas to extend and improve existing method specialization techniques.

## 1.4 Outline

Chapter 2 gives descriptions and examples for all the definitions that are used throughout the rest of the thesis. Chapter 3 gives an overview of the implementation of the Jikes Research Virtual Machine. Chapter 4 gives an overview of method specialization and how it is related to other optimizations. Chapters 5, 6, and 7 give a complete overview of customization, selective specialization, and automatic program specialization respectively. Each chapter gives an example showing the transformation of the optimization, along with a description and discussion of the results of the optimization. Chapter 8 discusses our new method specialization compiler optimization framework designed for the Java programming language. Chapter 9 lists and discusses the results we obtained through experimental testing. Chapter 10 gives a description of work related to devirtualization of method call sites. Chapter 11 gives the conclusion we can draw from previous research and the future direction of programming languages. Finally chapter 12 lists future work related to

method specialization research.

Appendix A is a listing of the raw numbers we obtained from our experimental testing and appendix B gives trademark information.

# Chapter 2

# Definitions

**Message Send** Message sending is used by object-oriented programming languages to allow polymorphism when performing method dispatch. Method dispatch involves sending a message to the class of the receiver-object type. That class, in turn, executes a target method. Because at runtime variables can hold different types of objects, the target method of a single method dispatch can be different at different times in the same execution.

**Receiver Object Type** When performing message sending, the target class depends on the class-type of the object receiving the message. The class of the receiving object is called the receiver object type. A function call is *monomorphic* or *static* if it has only one receiver type, and *polymorphic* or *virtual* if it has more than one receiver type.

Control is transferred between functions through function calling. A *caller* method contains a *call site* that specifies the transfer of control to a *callee*. For instance, in Figure 2.1 method *main* calls method *foo*. In this example *main* is the caller, *foo* is the callee, and the call site for *foo* is the last statement in *main*. A method may contain several call sites with the same callee.

**Formal Parameters and Actual Parameters** A formal parameter is a parameter that appears in the method declaration. A formal parameter must be a variable. In Figure 2.2, method declaration *foo(Integer, Integer)* contains two formal parameters: variable *a* and variable *b*. Method *bar(Integer)* contains one formal parameter, variable *x*. An actual parameter is a parameter given in a method call site. Actual parameters can be variables, constant values, method call sites, or expressions. In Figure 2.2 the call site *foo(var1, new*

```
1. public class A {
2.   public static void main(String[] args) {
3.         foo();
4.         bar();
5.         return;
6.   }

7.   private void foo() { ... }

8.   private void bar() { ... }
9. }
```

Figure 2.1: A Sample Pseudo Java Class Hierarchy

*Integer(3))* has two actual parameters: first the variable *var1*, and second the method call site *new Integer(3)*. Additionally, in method *foo(Integer, Integer)* the call sites *bar(a)* and *bar(b)* include actual parameters variable *a* and variable *b*, respectively.

```
public class A {
    public static void main() {
        Integer var1 = new Integer(5);
        foo(var1, new Integer(3));
    }
    void foo(Integer a, Integer b) {
        b = new Integer(10);
        bar(a);
        bar(b);
    }
    void bar(Integer x) {
        ...
    }
}
```

Figure 2.2: A Sample Pseudo Java Program Demonstrating Different Parameters

**Pass-Through Call Site** A Pass-Through Call Site is a call site where the formal parameters are passed directly as actual parameters [29]. An example of a pass-through call site is found in Figure 2.2. Method *foo(Integer, Integer)* contains the call site *bar(a)* which is a pass-through call site because variable *a*, an actual parameter, is also a formal parameter to method *foo()* with no modification in-between. However the call site *bar(b)* is not a pass-through call site because variable *b*, though it is a formal parameter to method *foo(Integer, Integer)*, is modified before being used as an actual parameter.

**Weight** Weight refers to the run-time execution count of a call site, *i.e.* the number of times a particular call site is executed during a program run. Using Figure 2.1, if the call site to method *foo()* (found on line 3) in Figure 2.1 is executed just once, then it has a weight of 1.

**Static-typed Programming Languages** Static-typed languages, such as Java, C, and C++, require that the object type of variables be known at compile time and therefore these types must be declared before they are used. Static-typing increases type safety and allows for more optimization because the compiler knows the types at compilation time.[1] However static-typing also makes these languages more tedious as the syntax is stricter than dynamic-typed languages.

**Dynamic-typed Programming Languages** Dynamic-typed programming languages, such as Smalltalk, Python, and Ruby, do not require the object-types of variables to be declared at compilation time. These types are determined at run time. As a consequence, the syntax of the language is more flexible. However, this flexibility impedes many optimizations at compilation time.

**Basic Block** A basic block is an ordered sequence of consecutive instructions that has the following properties:

- The flow of control can only enter at the first instruction in the block of instructions.
- The flow of control can only leave at the last instruction in the block of instructions.

Therefore, if one instruction of a basic block is executed, then all instructions in the block must also be executed.

**Extended Basic Block** An extended basic block is similar to a basic block as it is an ordered sequence of consecutive instructions and the flow of control can only enter at the first instruction. However in an extended basic block, any instruction can be an exit point from the block. Thus an extended basic block has a single entry point, but multiple exit points [97].

[1] This assumption is only partially true for Java because Java provides dynamic class loading.

**Control Flow Graph** A Control Flow Graph (CFG) is a directed multi-graph representing the flow of control through a program. Each node in a CFG refers to a basic block and each directed edge between nodes represents flow of control from one node to another. An edge exists from node B1 to node B2 if:

1. The last instruction of B1 is a jump instruction to the first instruction of B2; or
2. The last instruction of B1 is not a jump instruction, and B2 immediately follows B1 in the program code.

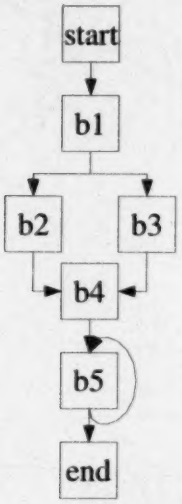


Figure 2.3: A Control Flow Graph

Figure 2.3 shows an example of a control flow graph. Every CFG has a single start block and a single end block to indicate where execution control can enter and exit. Block *b1* has 2 exits from it indicating that program execution can jump to either block *b2* or *b3* but not both. Block *b4* has two entry points indicating that control can enter *b4* from either block *b2* or block *b3* but not both.

**Program Call Graph** A program call graph $G = (N, E)$ is a directed multi-graph with a set of $N$ nodes that represent whole methods, and a set of $E$ edges that represent call sites [16]. An edge from node $N_1$ to $N_2$ indicates that a call site

in method $N_1$ has method $N_2$ as the target method. Because there are several call sites within a method, there can be several outgoing edges from a node. Likewise because a method can be the target of several call sites, a node can have multiple incoming edges.

There are two types of program call graphs:

- A *Static Call Graph* is a call graph created offline that contains edges for all possible targets of each call site. Therefore if a call site is polymorphic, an edge is given for each potential target class.
- A *Dynamic Call Graph* is a call graph created dynamically at run time. The graph only records edges for target classes of call sites during the program execution. Additionally a weight can be assigned to each node and edge where node weights indicate the number of times a method is called, and where edge weights indicate the number of times a call site with the corresponding target class is executed. Figure 2.4 shows a sample dynamic call graph with edge weights.

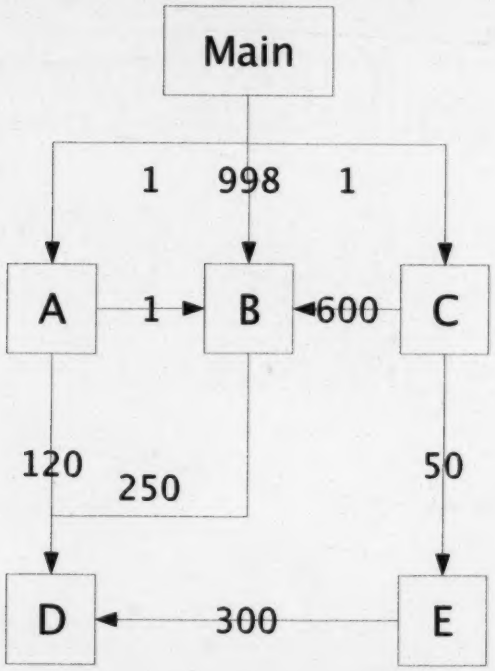


Figure 2.4: A Program Call Graph

**Virtual Method Table** A virtual method table, also known as a virtual table or a vtable, is a data structure used to facilitate run-time method lookup. The table contains pointers to the corresponding method implementations for the class. When a message is sent to an object, the program uses the virtual method table to determine which method to call. A virtual method table is the same for all objects in a class. Thus, most implementations share a single version for each class.

```
public class A {
    void func() {
        foo();
        bar();
    }
    void foo() { ... }
    void bar() { ... }
}

public class B extends A {
    void foo() { ... } // Overridden method from Class A
    void bar() { ... } // Overridden method from Class A
}

public class C extends B { ... }
```

Figure 2.5: A Sample Pseudo Java Class Hierarchy

Figure 2.5 gives an example of a class hierarchy in Java pseudo code, and Figure 2.6 shows the corresponding virtual method table.

**Static Call Sites** Static Call Sites (also known as *Monomorphic* call sites) refer to instructions that transfer control to a method where there is only one possible target method. Additionally, because the target method is known at compilation time, the compiler may perform inter-procedural optimizations.

**Virtual Call Sites** Abstraction is an object-oriented technique used to hide the details of an object's implementation from the object's clients [4]. Abstraction allows for the receiver object of a call site not to be known until run time. This occurs when the dispatch of the method depends on the dynamic type of the receiver object. A call site whose receiver is determined at run time is called a virtual call site.

Virtual call sites inhibit compiler optimizations because the exact target method is not known at compilation time. Optimizations, such as inlining and inter-

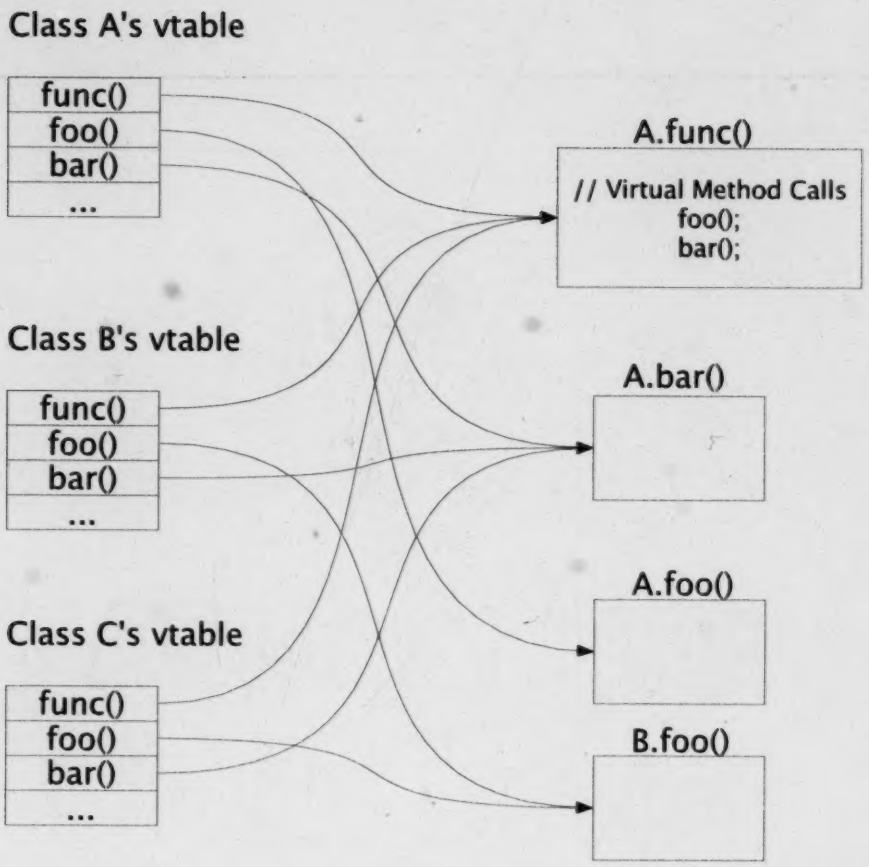


Figure 2.6: A Virtual Method Table For The Class Hierarchy In Figure 2.5

procedural analysis, cannot be performed if the target method is not known. In addition, virtual call sites negatively impact branch prediction [25, 55, 110] because it is difficult to predict where the program will go.

**Applies-to Set** An applies-to set [29] is a set of classes where a method implementation is the target of a method dispatch. For example, the applies-to sets for the *foo()* method in the hierarchy of Figure 2.7 are shown in Figure 2.8.

```
public class A {
    void func() {
        foo();
    }
    void foo() { ... }
}
public class B extends A {
    void foo() { ... }
}
public class C extends B { ... }
public class D extends C {
    void foo() { ... }
}
public class E extends C { ... }
public class F extends A { ... }
```

Figure 2.7: A Sample Pseudo Java Class Hierarchy

The class hierarchy for the code in Figure 2.7 is shown in Figure 2.8. This hierarchy is divided into three sections according to the reachability of method *foo()*. If a method call to *foo()* happens anywhere within a hierarchy section, the method dispatch would be to the top class in the section.

**Program Context Change** Long running programs often execute a small portion of the program's code for an extended period of time. Sometimes the program execution changes to execute a different portion of the program's code for another extended period of time. This change from one portion to another is called a program context change or *program phase change*.

Context changes may also include data values that are used for an extended period of time and then changed to a different value in the same execution instance.

**Method Call-Site Devirtualization** Method call-site devirtualization is the process of identifying virtual call sites that have a single target method and then

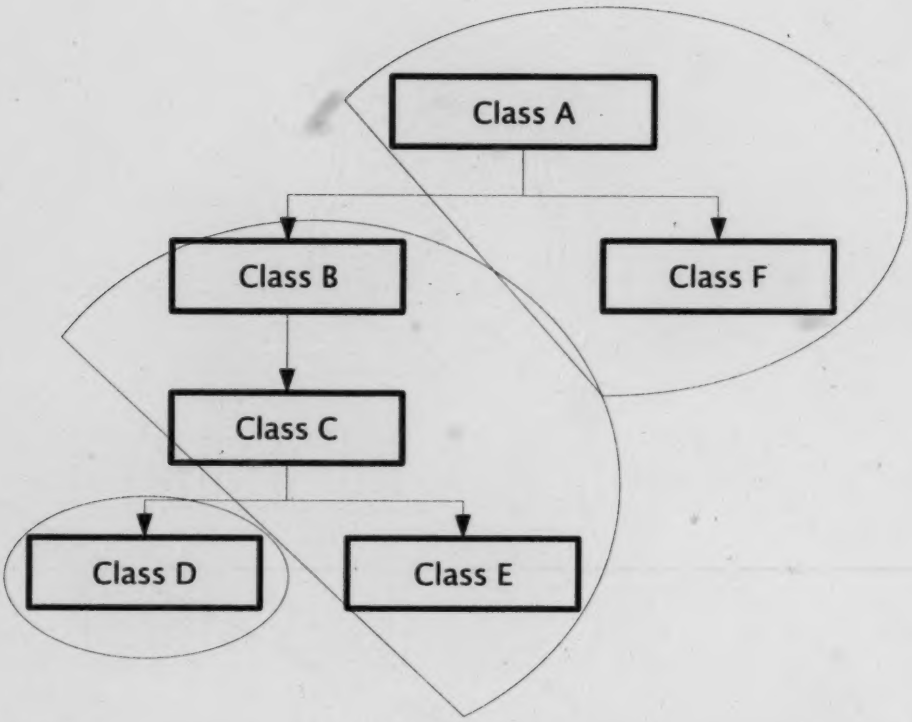


Figure 2.8: The Applies-To Sets For Method *foo()* For The Class Hierarchy In Figure 2.7

changing virtual call sites into static call sites. Several compiler optimizations have been introduced to devirtualize call sites [4, 25, 50, 51, 65, 76, 118]. While devirtualizing a call site eliminates a method dispatch, the main benefits come from creating opportunities for inter-procedural optimizations such as inlining [76].

**Profiling Information** Profiling is the process of collecting various pieces of information from a running program to predict the future execution of the program. The process of inserting instructions into code to collect profiling information is known as *instrumenting* the code. Collected profiling information can vary from the value of particular variables, control flow along the control flow graph edges, or the frequencies of runtime receiver types at call sites.

Collected profile information can then be used by the compiler to make better decisions for the optimizations that it performs, such as inlining [15], receiver class prediction [65], code reordering [104], instruction scheduling [95], and other optimizations [35]. Optimizations that utilize profiling information are called *feedback directed optimizations.*

Profiling information may be collected offline or online:

- Offline: Profiling information is collected by performing a training run of the program which stores the profile information into a file that can later be accessed and used by the compiler to recompile the program using feedback-directed optimizations. The advantage to this method is the ability to gather profiling information before the release of the program at no runtime cost.

  Offline profiling, also called *static profiling*, requires one or several runs of the program with "typical" input data. The accuracy of the information collected by offline profiling depends on the correlation between the program inputs selected for profiling and the actual inputs for a program run. If the profiling input does not reflect the actual runtime program behavior, then offline profiling may hurt the program performance.

  Additionally, offline profiling does not allow a virtual machine to adapt to program execution context changes because the optimizations rely on a single data set to predict the program behavior.

- Online [19]: Profiling information is collected during the execution of the program, allowing for better data collection and also allowing virtual machines to detect and adapt to context changes. However as the executed code is instrumented with instructions to collect profile data, performance degradation can be in the range of 30% to 1000% [19].

  As a result of this performance hit, sampling techniques have been introduced to minimize the overhead caused by instrumented code [19]. Sampling involves copying the basic blocks that are to be instrumented, and instrumenting one set of the basic blocks while leaving the other set untouched. There are 2 techniques used to switch between instrumented and original basic blocks:

  - Time-Based Sampling: At specific time intervals, the virtual machine interrupts the program and tells it to begin executing the instrumented code. The instrumented code is then executed for a specific period of time before the virtual machine tells the execution to start executing the un-instrumented code again. This is generally done at the end of basic blocks to make switching back and forth easier.
  - Counter-Based Sampling: Checking code is inserted at loop entries and loop back-edges in the original code. This checking code includes counter increment instructions and checks to see if the counter has reached a threshold. When the counter reaches the threshold, it is reset and control is transferred to the instrumented code. The instrumented code then runs through until a loop-backedge or a new basic block is encountered, where control returns to the original code.

**Receiver Class Distribution** Receiver class distribution is a type of profiling information that gives the frequency of types for receiver objects at call sites [29]. This information is especially useful for method specialization because it allows the compiler to predict which class type receives the majority of method calls.

A sample receiver-class distribution for the call site *foo()* in method *A.func()* for the example of Figure 2.7 is given in Figure 2.9.

Figure 2.9 shows that the receiver-class distribution is peaked towards class *B*. Also note that even though class *C* receives 100 method dispatches for method

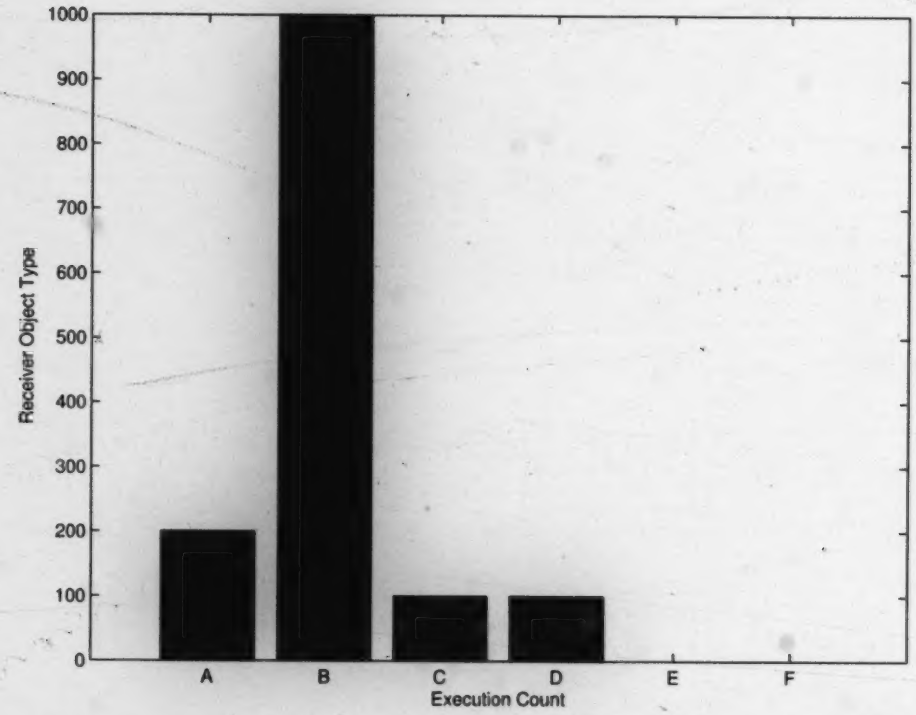


Figure 2.9: A Receiver-Class Distribution For the Call Site *foo()* in Method *A.func()* from Figure 2.7

*foo()*, the actual target method is *B.foo()* due to the method reachability of *foo()* (See Figure 2.8).

**Dynamic Class Loading** Dynamic class loading is a process that allows a class to be loaded into the class hierarchy at any point in the program execution. The dynamic loading of a class only requires the name and location of the class. Dynamic class loading allows development of easily extendable software. Developers can write code that install components at run-time and upgrade software components without having to restart the program.

In Java, there are 2 ways to dynamically load a new class:

1. *The Class.forName(String) method* [100]: This method takes a class name as a string parameter, loads it into the program, and returns the Class object.
2. *User-defined class loaders* [88]: User defined class loaders allow the programmer complete control over how classes are loaded, including specifying remote locations to find classes, checking security of classes that are loaded, reloading classes, and even instrumenting classes.

**Method Inlining** Inlining [5, 13, 16, 51, 69, 122] is a program transformation that replaces a call site with the instructions of the callee. Inlining removes the overhead of performing method lookup and linking. It also increases the scope for intra-procedural optimizations such as constant folding, common subexpression elimination, global register allocation, and others.

In Figure 2.10 the inlining of method *getSomeInt()* in method *func()* would be desirable. Figure 2.11 shows *func()* after inlining. The call site *getSomeInt()* is replaced with the code of the method *getSomeInt()*. This creates an opportunity for constant propagation in the following instruction.

Inlining requires that the call site be monomorphic, however there are techniques to get around this requirement such as guarded inlining [13]. Guarded inlining is used with programming languages that allow dynamic class loading, where the optimization finds call sites that are monomorphic at the current point in the program execution. Upon finding a call site that is currently monomorphic and that the compiler selected for inlining, the compiler guards the call with a conditional statement that checks to see if dynamic class load-

```
void func() {
    int a = getSomeInt();
    int b = a;
    ...
}

int getSomeInt() {
    return 5;
}
```

Figure 2.10: Sample Java Pseudo Code Before Method Inlining

```
void func() {
    int a = 5;
    int b = a; // Opportunity for constant propagation
    ...
}

int getSomeInt() {
    return 5;
}
```

Figure 2.11: Sample Java Pseudo Code After Method Inlining

ing has taken place. If dynamic class loading has not taken place, control flows to the inlined call site. Otherwise control flows to the original call site that could now be polymorphic.

An adverse effect of inlining is an increase in the size of the executable code. This code-size growth can cause side effects such as an increase in the number of instruction cache misses and an increase in compilation time. Thus most compilers implement an inlining heuristic to decide which call sites to inline.

**Static-Compilation Systems** Static compilation systems are systems that perform compilation of instructions off-line, or before program execution. An example would be the GCC compiler suite [1]. Static compilation systems have no run-time cost.

**Way-Ahead-Of-Time Compilation Systems** Way-Ahead-Of-Time (WAT) compilers [106] are static compilation systems used to compile Java source code directly into machine code. While WAT offers substantial performance benefits by allowing optimizations to be performed off-line with no run-time performance hit, it forfeits the portability of the Java classfile along with dynamic

class loading.

**Dynamic-Compilation Systems** Dynamic compilation systems, also known as *Just-In-Time* (JIT) compilers, are systems that perform compilation on-line or during program execution. Thus the time taken to compile instructions is part of the total time taken to execute the program. However these machine instructions can then be cached and reused without the cost of having to interpret the bytecodes on each execution. The Jikes Research Virtual Machine [44] is an example of a dynamic-compilation system.

**Recompilation Systems** A recompilation system is a dynamic compilation system that performs incremental compilation, or compiles methods at different optimization levels, generally using online profiling data to make decisions. Usually a compiler has predefined levels that contain certain optimizations, where the lower levels contain easier, less time-consuming optimizations and the higher levels contain more computationally intensive optimizations. The compiler uses online-profiling data to select which level to compile a portion of code (usually a whole method).

For example, a typical recompilation scheme in a Java virtual machine compiles a method at level 0, with little or no optimizations, the first time the method is executed. After the program has been executed for a certain period of time, the profiling information collected is examined to determine how often the method has been executed over the defined period. If the method qualifies for a higher level of compilation, the compiler recompiles the method at the new optimization level, and replaces the old method with the newly compiled method. This process can be repeated until the method is compiled at the highest compilation level offered by the compiler.

**Class Hierarchy Information** Object-oriented languages allow techniques such as inheritance and dynamic binding of method call sites to make code easier to reuse, extend, and design. Class hierarchy information is a collection of all of a program's class hierarchy inheritance information. When a class $C$ is compiled, the compiler knows that $C$ is a subclass of some other class and therefore inherits the methods defined by it's superclass along with potentially overriding some of these methods.

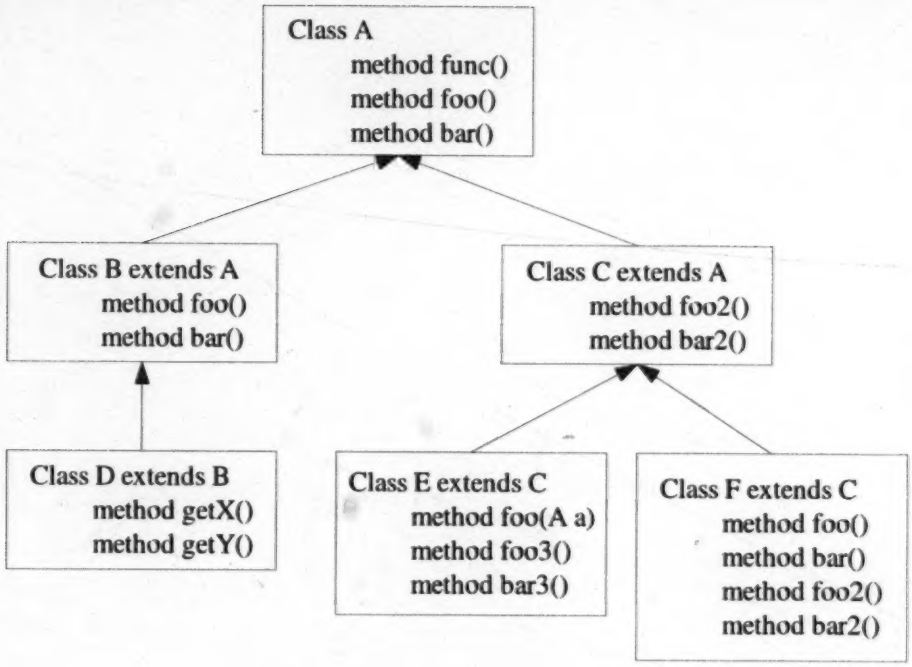


Figure 2.12: A Class Hierarchy Graph

Figure 2.12 shows a sample class-hierarchy graph. The nodes in a class-hierarchy graph represent classes, and the edges represent "is a subclass of" information. In this example classes $B$ and $C$ are subclasses of class $A$ . The figure shows which methods are defined in each class. By following the edges in the graph, the compiler can determine which methods override already defined methods. For example, class $B$ overrides methods *foo()* and *bar()* from class $A$ while class $D$ defines 2 new methods *getX()* and *getY()* but still is able to access the inherited methods *B.foo()* and *B.bar()*.

Class hierarchy analysis is used to expose opportunities for call site devirtualization [50]. Once class hierarchy information has been collected, the compiler can correlate call sites with the class hierarchy to determine when there is a single applicable target method to a call site. Such call sites are called *monomorphic* and can be devirtualized.

Since Java allows for dynamic class loading, the class hierarchy information for a Java program is never complete. Dynamic class loading allows for a class to be loaded into the class hierarchy at any point in the execution of

a Java program. Thus the class hierarchy information available for a Java program must be associated with a point in the program execution. Java classes cannot be removed from the class hierarchy once they are loaded by the virtual machine.

**On-Stack Replacement** On-stack replacement, also called *dynamic de-optimization*, is a technique that was originally used for debugging globally optimized code [74]. Debugging requires functionality such as single-stepping through source code and the ability to change variable values at any execution point. Optimizations, such as constant propagation and folding, may remove unnecessary instructions in optimized code. Since these instructions are needed by debuggers to allow the user to step through code and make modifications in the middle of execution, a debugger must be able to de-optimize code.

While on-stack replacement was originally designed to allow debuggers to de-optimize code, it has also been used for speculative optimizations such as inlining in the presence of dynamic class loading. For example, a method call site can be speculatively inlined. If a newly loaded class invalidates the optimization, the method can be de-optimized and replaced by the correct, unoptimized, code.

The cost of performing on-stack replacement is high and requires several data structures to record not only the optimizations that have been done, but also how to de-optimize them. On-stack replacement is a complex technique to implement and thus not many JVMs currently implement it.

**Multi-Methods** When method dispatch is based on a single argument to the method — usually the type of the receiver object — then it is called *single-dispatch*. Single-dispatch is implemented in C++, Smalltalk, and Java. If method dispatch depends on the run-time type of 2 or more arguments, then it is called *multi-dispatch*. Multi-methods are methods that utilize multi-dispatch. MultiJava [2, 40] is an extension to the Java programming language that adds multi-dispatch functionality to Java.

**Aspect-Oriented Programming** Aspect Oriented Programming [47, 58, 60, 61, 82, 83, 84, 99, 127] is a programming style that enables *separation of concerns* in object-oriented programming languages. This separation of concerns

prevents the construction of *tangled* code (code that is spread across multiple classes).

An *aspect* is defined as a modularized unit of crosscutting concern. A crosscutting concern occurs when object-oriented modularization (classes) is unable to effectively define a programming concern in a single unit .

Aspects allow the programmer to design concerns in a modular, organized fashion. Some examples of concerns that may be coded as aspects include:

- High-level concerns such as security
- Low-level concerns such as caching
- Functional concerns such as business rules
- Non-functional concerns such as synchronization

AspectJ [83] is an aspect-oriented extension to the Java programming language. The following examples in Figures 2.13, 2.14 and 2.15 are based on the example found in [83]. AspectJ adds the following aspect-oriented features to Java:

- *Pointcuts*: A *Join Point* is a well-defined point in the execution flow of a program. A pointcut is a set of join points.

  Examples of join points include: method calls, setting the value of a field, getting the value of a field, exception handler executions, etc. An example of a pointcut, named *move* is displayed in Figure 2.13.

```
pointcut move() :
    call(void Point.setX(int)) ||
    call(void Point.setY(int));
```

Figure 2.13: An Example Pointcut, After [83]

  This pointcut is executed whenever either method *Point.setX(int)* or method *Point.setY(int)* are called.

- *Advice*: An advice defines the instructions that are executed at the join points specified by a pointcut. An advice may be executed before (immediately before the target point), after (immediately after the target point), or around (in place of the target point) pointcuts.

```
after() : move() {
    Display.update();
}
```

Figure 2.14: An Example of Advice, After [83]

Figure 2.14 shows an advice that executes the block of code whenever the execution reaches a point as defined by the pointcut *move* (See Figure 2.13). When execution reaches that point, the code in the block is executed. In this case, the display is updated by calling the *Display.update()* method.

- *Aspects*: Aspects combine pointcuts and advice into a modular unit.

```
aspect Movement {
    pointcut move() :
        call(void Point.setX(int)) ||
        call(void Point.setY(int));
    after() : move() {
        Display.update();
    }
```

Figure 2.15: An Example of an Aspect, After [83]

Figure 2.15 shows a complete aspect that executes display update instructions whenever the location is changed.

# Chapter 3

# Overview of The Jikes Research Virtual Machine

The Jikes™ Research Virtual Machine (RVM) [6, 8, 10, 24, 44] is an open-source virtual machine for Java written in Java. The project started in December of 1997 at the IBM T. J. Watson Research Center as a study to determine the practicability and desirability of implementing a JVM in Java. Initially this RVM was called Jalapeño Virtual Machine. Jikes currently runs on IA32/Linux and PowerPC/AIX Unix architectures.

This chapter gives a brief overview of the aspects of the Jikes RVM that are most relevant to this thesis.

## 3.1 Booting Jikes

Because Jikes is written in Java, in order to begin execution a boot-image of core-essential services such as a class loader, an object allocator, and a compiler are written into a file. This file is then loaded into memory and executed to load the necessary classes needed to support execution. Once this is done, the multi-threading subsystem is initialized by creating virtual processors on top of operating system threads, thus allowing Java threads to be multiplexed. Jikes implements a preemption thread system that allows threads only to be preempted at predefined *yieldpoints*. These yieldpoints are inserted at method prologs (see Section 3.4) and loop back-edges. Finally a Java thread is created to run the program specified in the command line parameter.

## 3.2 Object and Memory Layout

Jikes uses an object model that allows fast virtual method resolution along with fast array and field accesses. The design uses an object header that consists of 2 words: a *Type Information Block (TIB)* and a *status*.

The status word contains 3 bits that are responsible for:

- Controlling blocking
- Holding the default hash value of the object
- Holding information for the memory management subsystem

The TIB refers to an array of information about the object's class, including it's superclass, the interfaces it implements, offsets of fields, etc. The TIB also includes a virtual method table to perform virtual method lookup. A graphical representation of TIBs is shown in Figure 3.1.

Jikes includes a single array called the *Jikes Table Of Contents (JTOC)* containing references to all static methods as well as to all static fields. A reference to this array is kept in a dedicated register in the PowerPC architecture, however on IA32 the JTOC register is cached in a data structure pointed to by another dedicated register as the IA32 has fewer registers compared with PowerPC.

## 3.3 Memory Management Subsystem

The Jikes Memory Management Subsystem implements an abstract memory management to allow switching memory allocation schemes. Currently Jikes includes support for the following garbage collection schemes:

- SemiSpace (Copying)
- MarkSweep (Non-copying)
- GenCopy (Classic Copying Generational)
- GenMS (Generational with mark-sweep mature space)
- CopyMS (Non-Generational with copy/mark-sweep hybrid)
- NoGC (Allocation only, no garbage collection)

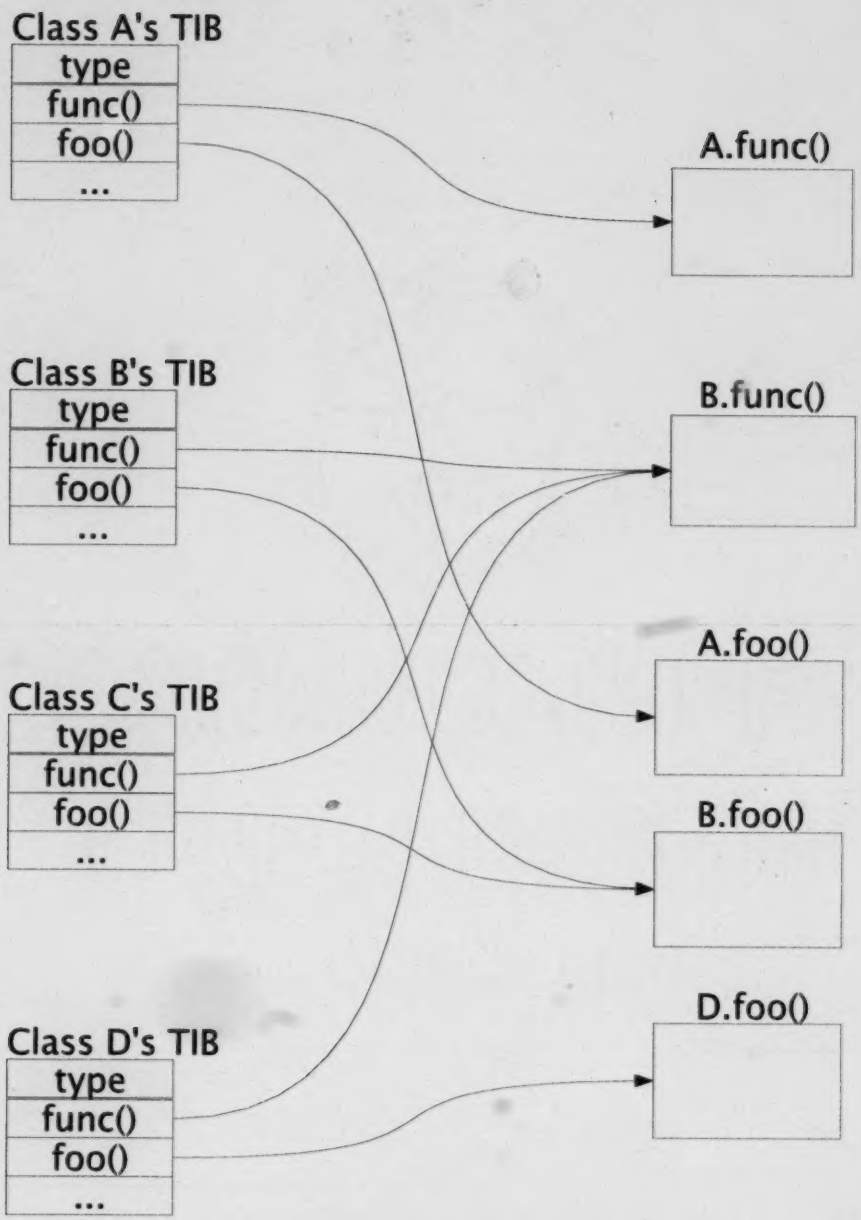


Figure 3.1: An example TIB

## 3.4 Intermediate Representation

The Jikes RVM compilers use a multi-level, register-based, intermediate representation (IR). Each IR instruction is an N-tuple, consisting of an *operator* and an arbitrary number of *operands*. Operators indicate the instruction to perform, while the operands are parameters to the instruction and are used to represent symbolic registers, physical registers, memory locations, constants, branch targets, method signatures, types, and others. The Jikes IR is closely related to Java bytecode. It uses Java specific operators and optimizations while preserving Java type information.

The Jikes IR is split into 3 separate levels:

- High-Level IR (HIR): Is a high-level representation that is similar to Java bytecode. However operators use symbolic registers instead of an implicit stack. HIR contains operators to implement checks for run-time exceptions.
- Low-Level IR (LIR): This level is more specific than HIR and includes details of the Jikes runtime and object layout (for example, addresses into the JTOC and TIBs). LIR also expands complex instructions, such as a TABLE_SWITCH instruction, into several smaller instructions.
- Machine-Specific IR (MIR): MIR is the most specific of the three levels. MIR is similar to the assembly code of the target architecture. Machine specific instructions are performed in MIR.

The compiler front-end converts Java bytecode into HIR. This front-end is broken into two pars: The BC2IR algorithm that translates bytecodes to HIR and an HIR optimizer that performs both on-the-fly optimizations as well as additional optimizations on the IR after translation. The BC2IR algorithm interprets the Java Bytecodes based on the Java bytecode specification provided by the Java Virtual Machine Specification [90]. BC2IR translates bytecodes into the corresponding HIR instructions. It also identifies extended-basic-blocks and performs on-the-fly optimizations such as copy propagation, constant propagation, register renaming for local variables, dead-code elimination, and inlining of short final or static methods. Even though these optimizations are performed again in later phases by the compiler, they are also applied at this time to reduce the size of the HIR and to reduce future compile time.

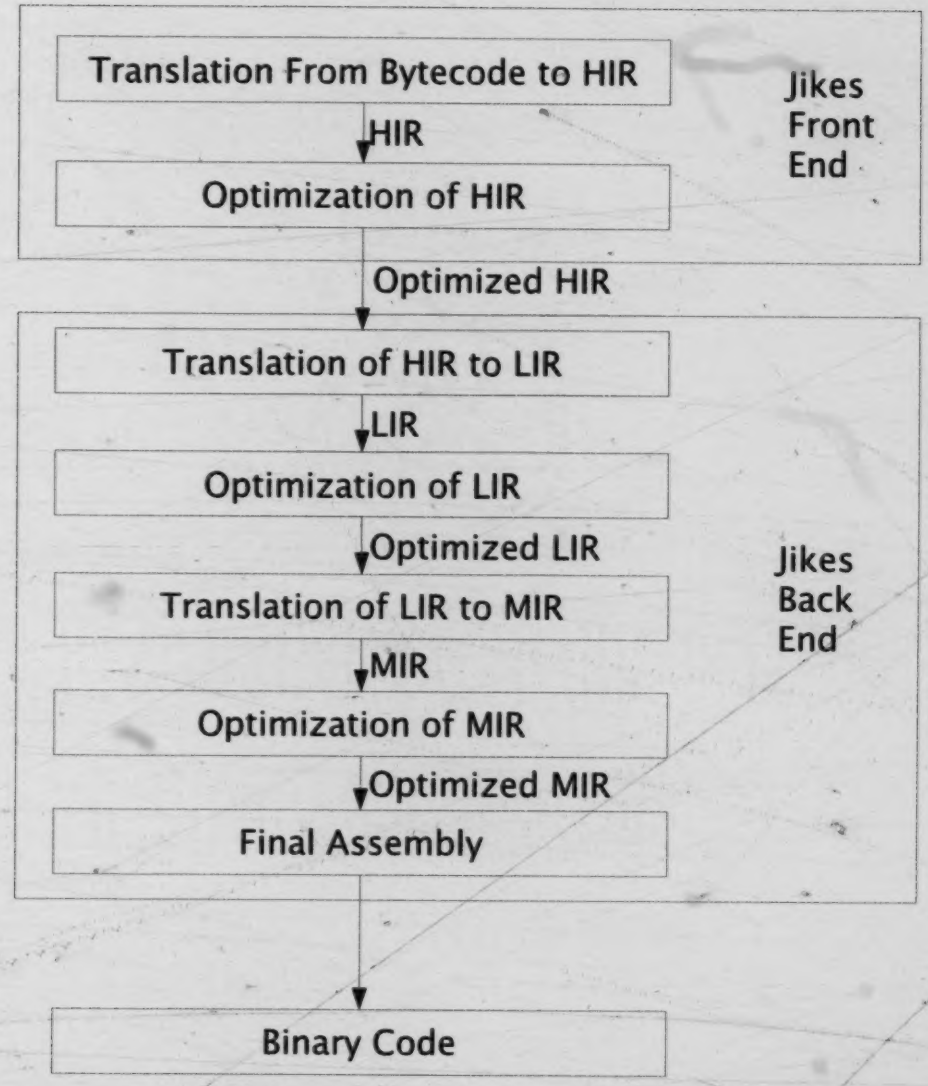


Figure 3.2: The Jikes RVM Intermediate Representation Transformation, after citeAlpern2000Jalapeno

After the HIR is generated, it goes through an optimization phase where simple transformations with modest compile-time overheads are performed. Generally the optimizations performed fall into 3 categories:

**Local optimizations:** (include but not limited to) Common sub-expression elimination; removal of redundant exception checks; and redundant load elimination.

**Flow-insensitive optimizations:** Copy propagation; dead code elimination; and conservative escape analysis.

**In-line expansion of method calls:** Guarded receiver type prediction.

The HIR is then converted into LIR instructions that are specific to the Jikes RVM. These LIR instructions contain information about object layout and parameter-passing mechanisms. For instance, an HIR instruction such as *invokevirtual* that is used to execute a virtual method, is expanded into 3 LIR instructions to:

1. Obtain the TIB pointer from the object.
2. Obtain the address of the appropriate method body from the TIB.
3. Transfer control to the method body.

Additional information is also generated, including a dependence graph for each extended basic block. This dependence graph includes the representation of true, anti, and output dependences for both registers and memory as well as control, synchronization, and exception dependences. This extra information causes the LIR to be two to three times larger than a corresponding HIR representation of a program.

The LIR then goes through an optimization phase. Currently only local common subexpression elimination is performed on LIR. In principle, any optimization performed on HIR can be performed on LIR but the size of LIR makes it less attractive than HIR for complex optimization algorithms.

The LIR is then converted to MIR where the dependence graphs for each extended basic block are partitioned into trees. These trees are then fed into the Bottom-Up Rewriting System (BURS) to produce the MIR. BURS uses an heuristic-based algorithm to choose instructions based on their cost. BURS allows for addi-

tional architectures to be easily added to the Jikes compiler by inserting the target instructions, selection rules, and costs into the system.

The MIR representation is then fed to the register allocation framework, that supports multiple allocation algorithms. The current scheme used is based on a greedy algorithm that does a linear-scan of the variables' live ranges. Method *prologs* and method *epilogs* are inserted at the start and finish of all methods. A prolog is used to:

- Save any nonvolatile registers needed by the method.
- Check if a yield has been requested.
- Lock objects if the method is synchronized.
- Act as a yield point.

An epilog is used to:

- Restore any saved registers.
- Deallocate the stack frame.
- Unlock objects if the method is synchronized.

The final phase is the assembly phase that emits the binary executable code of the method into an integer array. This phase also finalizes the exception table and converts intermediate-instruction offsets into machine-code offsets. These arrays are then stored into a field of the object instance for the method. Jikes also supports the storage of multiple versions of a method in an object, thus allowing methods compiled at different optimization levels or specialized methods to be stored in the object.

## 3.5 Compiler Subsystem

Like most modern virtual machines, Jikes uses a compile-only strategy. In other words, methods are compiled to native code upon their first execution. Jikes currently contains three separate compiler subsystems: A baseline compiler, an optimizing compiler, and an adaptive optimization system.

### 3.5.1 The Baseline Compiler

The Jikes baseline compiler is a simple compiler that generates code by simulating Java's operand stack. This baseline compiler does not perform register allocation. This compiler is provided as a reference and is expected to perform only marginally better than an interpreter.

### 3.5.2 The Optimizing Compiler

The Jikes optimizing compiler uses a compile-only approach that compiles methods at a predetermined optimization level when the method is first executed. The optimizing compiler makes use of the intermediate representation to perform it's optimizations. Currently the optimizing compiler contains three separate levels of optimization:

- Level 0 - Contains mostly optimizations that can be performed on-the-fly
- Level 1 - Augments Level 0 with local optimizations along with inlining that is based on static-size heuristics and global flow-insensitive optimizations.
- Level 2 - Augments Level 1 with SSA-based flow-sensitive optimizations.

### 3.5.3 The Adaptive Optimization System

The Adaptive Optimization System (AOS) [14, 15] implemented in Jikes is used to give more robust performance than a compile-only or an interpret-only approach.

Figure 3.3 was published in Arnold *et al.* [15]. It is reproduced in this thesis for the convenience of the reader. The Figure shows a graphical view of the adaptive optimization system and the interaction between the components. The following description of the Jikes adaptive optimization is based on [15] and related documentation.

The Adaptive Optimization System Architecture contains three subsystems: the *Runtime Measurements Subsystem*, the *Controller*, and the *Recompilation Subsystem.*

The Runtime Measurements Subsystem is responsible for gathering dynamic profiling data, organizing that data, and then passing it to the Controller for recompilation decisions. The Controller creates and controls units called *Organizers* that are used to process and analyze raw data at regular intervals. When these

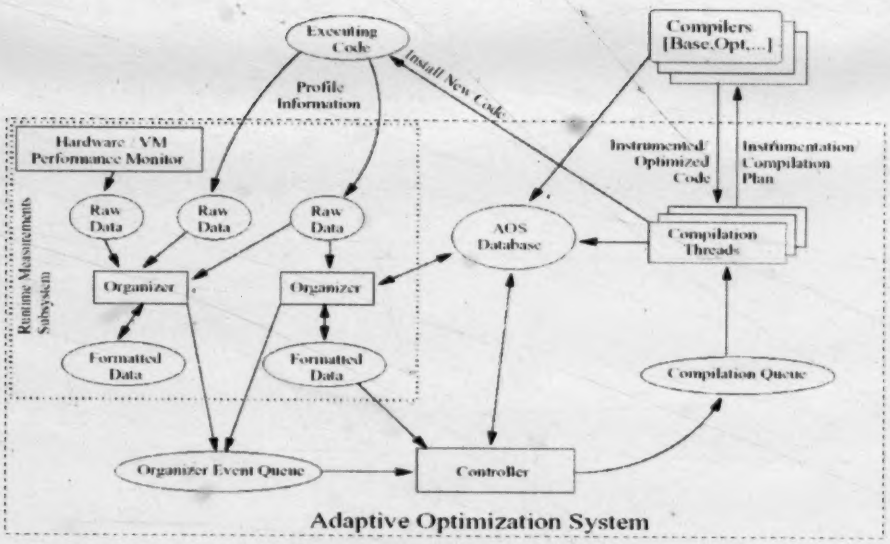


Figure 3.3: The Jikes Adaptive Optimization System as shown in [15]

organizers are issued by the controller to process data, they package the data into a form that can be conveniently used by the controller. Having organizers abstracted into their own class easily allows for gathering information for different uses from the same raw data. For example, different organizers can be created to monitor hot methods (for recompilation), or to monitor hot call edges (for inlining).

The profiling information collected follows the framework as described by Arnold *et al.* [19]. This framework creates duplicated code that contains instrumentation to collect the profiling information. Then, at specified intervals, the instrumented code gets executed for an assigned number of iterations, before control switches back to the un-instrumented code. This mechanism allows the framework to gather information without excessive intrusion on the program execution.

The Controller unit is used to make decisions about compilation and recompilation plans, and to query and to issue commands to perform these functions. The Controller coordinates operations of the Runtime Measurement Subsystem and of the Recompilation Subsystem. The Controller coordinates all activities to be performed by the Runtime Measurement System, including what kind of profiling information should be collected, for how long, and under what conditions subject to it. The Controller uses the information gathered to make decisions for the Recom-

pilation Subsystem. For example, given new profiling information, the Controller may decide to either stop collection of profiling information because it is no longer needed, or it may decide to recompile the method at a higher optimization level to increase execution speed.

The main function of the Controller is to use a cost/benefit analysis to determine whether a method should be recompiled at a higher optimization level in order to attempt to speed up program execution. The Controller makes these decisions for each compilation level using the following estimation process:

- $T_i$, The expected time the program will spend executing the method if it is not recompiled.
- $C_j$, The cost of recompiling the method at the new optimization level j.
- $T_j$, The expected time the program will spend executing the method, if the method is recompiled at level j.

The Controller then finds the level $j$ that minimizes $C_j + T_j$. If $C_j + T_j < T_i$, the controller recompiles the method at level $j$. Since there is no way to know how long a program will execute, Jikes currently estimates that the program will run for twice as long as it has already run. Jikes uses earlier profiling information to make an estimation as to how long the program will spend in a given method.

The recompilation subsystem contains compilation threads that are used to invoke the compiler. Since the compilation threads are separate from the application threads, recompilation can occur in parallel with running the application. Initial compilation must occur in application threads because for their first execution methods are compiled lazily. Compilation relies upon a compilation plan that is sent to the compiler. Each compilation plan is broken into three components: an optimization plan, profiling data, and an instrumentation plan. The optimization plan contains a list of the optimizations to be performed during compilation. The profiling data is used for decisions by feedback-directed optimizations. Finally, the instrumentation plan specifies instrumentation that needs to be inserted into a newly compiled method. Once a method is compiled, the compilation thread installs the new method in the JVM to be used in future executions.

The AOS database is used by all three subsystems as a central repository of information regarding decisions, events, and results. This database can be queried

by any system to obtain information for future decisions. An example of how this database is used involves the Controller recording it's compilation plans for method recompilations to track the status and history of past recompilations.

## 3.6 Inlining

Jikes uses inlining in both it's optimizing compiler and it's adaptive compiler subsystems. The next two subsections provide an overview of both:

### 3.6.1 Optimizing Compiler Inlining Overview

The inlining implementation used by Jikes abstracts inlining procedure and inlining policy into separate classes to allow for easy addition/extension of policies and techniques. Inlining policies are located in an Inlining Oracle object that is queried by the compiler to determine if a call site should be inlined.

The static Jikes optimizing compiler uses static size heuristics to make inlining decisions. The heuristics used consists of three rules:

1. The compiler first analyses the costs involved for the callee:
   - The cost of making a method call (a higher cost results in better gains for inlining).
   - The cost of a guard (this cost depends on the type of guard used).
   - The estimated size of the method after it is inlined (this size must be below a constant value) .
2. The depth of inlining is checked against a constant (currently 5) to make sure that recursive inlining does not go too deep.
3. The total amount of inlining done so far in the program to prevent excessive code growth.

### 3.6.2 Adaptive Compiler Inlining Overview

The adaptive compiler gathers profiling information for adaptive inlining by creating organizers to analyze the call stack and record the caller, the call site, and the callee in method prologues. This data is then inserted into organizers that are used to identify edges in the dynamic call graph where the number of samples taken surpasses a constant value threshold. These hot edges are then inlined only if the

method is recompiled and the size constraints are not broken. The edge hotness threshold used by the adaptive system starts out larger at program startup and then is reduced until it reaches a constant value in order to prevent a large amount of inlining to be performed at program startup and to allow a sufficient amount of profiling data to be collected.

The organizer also has the responsibility of identifying methods that are candidates for further recompilation to allow for inlining of hot call edges. The organizer uses estimation to determine the efficacy of the optimization given the new rules for re-optimization. The controller uses this information to make recompilation decisions. This estimation is the result of several different areas including removal of method call instructions and additional optimizations that can be performed as a result of inlining.

The AOS database stores inlining decisions and events. This database allows for the adaptive system to identify call graph edges that shouldn't be identified as candidates for inlining when the adaptive system has all ready refused to do so, and to keep track of previous inlining decisions so that recompilation doesn't lose these inlining candidates.

# Chapter 4

# An Overview Of Method Specialization

Object-oriented programming is often used by programmers for it's inheritance, encapsulation, polymorphism, and other advantages. Nevertheless, these advantages can cause the runtime execution speed of object-oriented programs to be much slower than their procedural programming language counterparts, without execution enhancing optimizations from compilers.

One of the reasons for this execution slowdown in object-oriented programming languages is due to dynamic dispatch or runtime method resolution. At a call site, the method implementation is dependent on the runtime type of the receiver object. Therefore, often the target method cannot be determined at compile time. As a consequence, run-time method lookup computations must take place when the call site instruction is executed. Besides the slowdown caused by the method dispatch, virtual call sites also slowdown the program by preventing compiler optimizations. For example, Calder *et al.* noted that indirect function calls cause unpredictable changes in program control flow. These calls may disrupt both instruction pipelines and speculative execution and thus cause cache misses [25].

For instance, the sample class hierarchy given in Figure 4.1 contains the call site *foo()* in method *A.func()* . Method resolution for this call site must be done at run time because there are three possible target methods at this point in the program execution. Methods *A.foo()*, *B.foo()* and *D.foo()* are potential targets. The actual target method cannot be known until the receiver object-type is know, which can only be determined at run time. Once this type is known, calculations can be made to determine the location of the correct method, and then control can be transferred to that location. Even though the calculation takes place at run time,

```
public class A {
    A localVar;
    void func() {
        foo();
    }
    void func2(A param) {
        param.foo();
    }
    void func3() {
        localVar.foo();
    }
    void foo() { ... }
}
public class B extends A {
    void foo() { ... } // Overridden from class A
}
public class C extends B { ... }
public class D extends C {
    void foo() { ... } // Overridden from class B
}
public class E extends C { ... }
public class F extends A { ... }
```

Figure 4.1: A Sample Pseudo Java Class Hierarchy Before Method Specialization

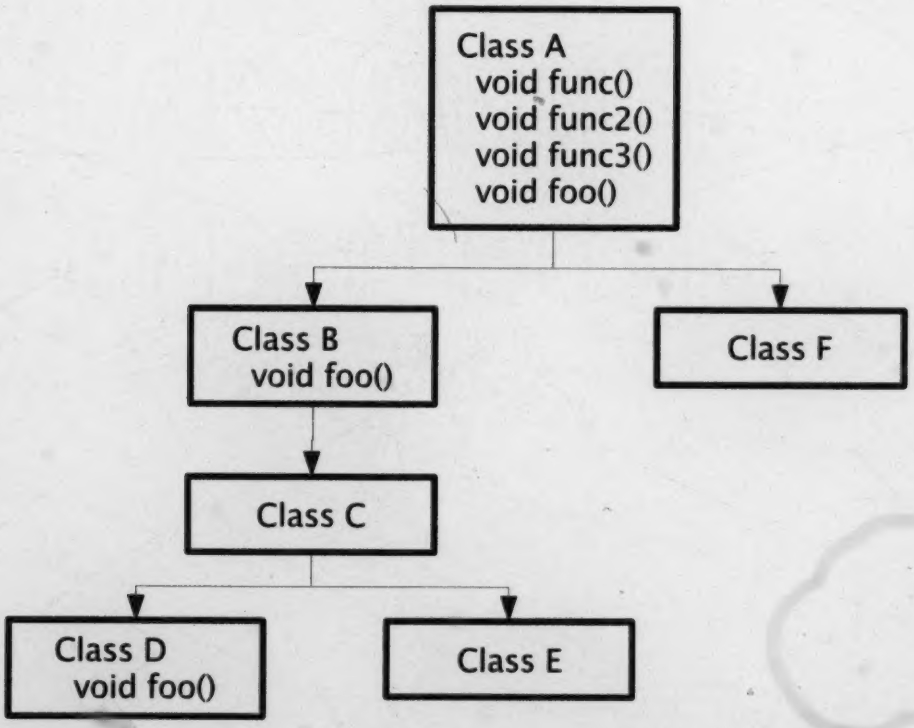


Figure 4.2: A Graphical Representation of The Class Hierarchy From Figure 4.1

modern compilers make method lookup cost insignificant [6].

Virtual call sites also slow down programs by preventing inlining. Inlining (see Section 2) is an extremely important optimization because it creates opportunities for other inter-procedural optimizations that may result in significant execution speedups. In the example of Figure 4.1, the receiver object type for the call site *foo()* is not known at compile time. Therefore *foo* cannot be inlined and, consequently, the statements in *foo* cannot be optimized along with the statements in its caller reducing potential speedup.

## 4.1 A Introduction To Method Specialization

One optimization created to expand the application of inlining in object-oriented programs is *method specialization* [26, 48, 109, 111]. Method specialization creates multiple versions of a virtual method. Each version is customized by selected characteristics to an applicable subset of classes. This versioning allows for more accurate static type information in the specialized method, which in turn may create opportunities for additional optimizations that would not take place before method specialization.

Method specialization can be implemented in several different ways. Programmers in the past have performed method specialization by hand by manually adding methods that contain more specific type information for highly-executed instances. However this method is tedious and error-prone, and resulted in development of automatic method specialization tools. Offline method specialization tools have been created that allow the programmer to express what they want to specialize through language extensions. A static compiler can then take that information and automatically generate a specialized program. Compiler optimization techniques have also be developed where no guidance is given from the developer, but rather the compiler itself makes the specialization decisions based on class hierarchy analysis and profiling information.

## 4.2 Method Specialization And Call Site Devirtualization

As object-oriented programs contain several virtual call sites, method specialization can be used to reduce the number of virtual call sites through devirtualization.

Method call site devirtualization is the process of identifying virtual call sites that only have one target method or changing virtual call sites into static call sites. Several compiler optimizations have been developed to devirtualize call sites [4, 25, 50, 51, 65, 76, 118]. While devirtualizing call sites provides benefit from the elimination of performing method dispatch, the main benefits come from creating opportunities for other optimizations such as inlining [76].

Consider the pseudo-Java class hierarchy in Figure 4.1. This figure shows how method specialization can be used to perform method call-site devirtualization. Here method *A.func()* contains a virtual call site, *foo()* . However, as shown in Figure 4.3, a class hierarchy analysis reveals that after the creation of specialized methods *B.func()* and *D.func()*, all *foo()* call sites in the *func()* method have only one possible target method, which allows the devirtualization of the call sites. This devirtualization is only possible in the absence of dynamic class loading.

The following examples will represent method devirtualization by appending the target classname on to the call site (example: *A.func()* ). When these examples are actually coded in Java, the bytecode compiler outputs a call site to a static *foo()* method, instead of the *foo()* instance method.

## 4.3 Method Specialization On Other Program Properties

Besides specialization based on the receiver type, method specialization can also be performed with respect to other properties of a program [109, 111]:

- *Data Encapsulation*: A method can be specialized with respect to the values of variables that it uses. This specialization allows for the elimination of memory references and may trigger additional optimizations.

  For example, consider a method *foo()* whose only parameter is an integer *a*. We may determine, either through static profiling or dynamic profiling that *a* is usually the value 5. Thus, a specialized version *foo5()*, where the instructions expect the value of *a* to be 5, may be created. Then, depending on the value of *a*, either *foo5()* is called (when *a* is 5) or *foo()* is called (when *a* is not 5).

- *Imperative Computations*: Methods can be specialized in relation to common compiler optimizations such as constant propagation, constant folding, conditional reduction, loop reduction, and others. For example, a method may

```
public class A {
    A localVar;
    void func() {
        A.foo(); // Devirtualized Call Site
    }
    void func2(A param) {
        param.foo();
    }
    void func3() {
        localVar.foo();
    }
    void foo() { ... }
}
public class B extends A {
    // New specialized method
    void func() {
        B.foo(); // Devirtualized call site
    }
    void foo() { ... } // Overridden from class A
}
public class C extends B { ... }
public class D extends C {
    // New specialized method
    void func() {
        D.foo(); // Devirtualized call site
    }
    void foo() { ... } // Overridden from class B
}
public class E extends C { ... }
public class F extends A { ... }
```

Figure 4.3: A Sample Pseudo Java Class Hierarchy After Method Specialization

```
public class A {
    int func(int a) {
        int b = a + 1;
        return b;
    }
}
```

Figure 4.4: A Sample Pseudo Java Method Before Data Encapsulation Method Specialization

```
public class A {
    int func(int a) {
        int b = a + 1;
        return b;
    }

    int specializedFunc(5) {
        return 6;
    }
}
```

Figure 4.5: A Sample Pseudo Java Method After Data Encapsulation Method Specialization

have an *if* statement that is based on a value set outside of the method. If the loop conditional value is usually true, a specialized method can be created which assumes that the conditional value is true, and thus eliminates the if statements (and possibly creates opportunities for more optimizations).

```
public class A {
    void func() {
        boolean test;
        ...
        if(test)
            bar();
        else
            foo();
        ...
    }
    void foo() { ... }
    void bar() { ... }
}
```

Figure 4.6: A Sample Pseudo Java Class Hierarchy Before Imperative Computation Method Specialization

## 4.4 Under-specialization and Over-specialization

When implementing method specialization, a strategy must be chosen in order to make decisions as to whether or not to specialize a particular method. This strategy should look to minimize 2 issues: under-specialization and over-specialization.

**Under-specialization** Deciding not to specialize methods that would result in considerable benefit is known as under-specialization. Examples of under-specialization include:

```
public class A {
    void func() {
        boolean test;
        ...
        if(test)
            bar();
        else
            foo();
        ...
    }
    void specializedFunc() {
        boolean test;
        ...
        bar();
        ...
    }
    void foo() { ... }
    void bar() { ... }
}
```

Figure 4.7: A Sample Pseudo Java Class Hierarchy After Imperative Computation Method Specialization

- Refusing to specialize a method due to space, time, or other concerns.
- Only specializing on a subset of potential candidates. For example, only specializing on the receiver object, when specialization of a method for a particular argument class could also result in significant gains.

**Over-specialization** Over-specialization is a result of performing method specialization too aggressively. Over-specialization may increase memory requirements, while causing no significant impact on execution speed. Since method specialization involves duplicating methods, memory requirements grow with the number of methods specialized. Examples of over-specialization include:

- Creating multiple specialized methods that are virtually identical when they could be coalesced into a single method.
- Creating specialized methods that will never be called or so rarely called that the cost of compiling the method is greater that any gain on execution speed due to that specialization.

## 4.5 Method Specialization Selection Strategies

Compilers that implement method specialization must choose a strategy to decide when to specialize methods. The strategy chosen may depend on several factors:

- Whether the compiler is a static-compilation system or a dynamic-compilation system. Static-compilation systems have the benefit that compilation time is not a part of execution time. The compiler can therefore choose a more aggressive method specialization strategy without affecting run-time performance. Dynamic-compilation systems require a more cautious strategy as run-time compiler cannot afford to spend as much time compiling methods unless the execution speedups are significant enough to justify the cost.

  Generally static compilation systems tend to implement a strategy that over-specializes the program because the extra compilation time is not noticed by the user. Dynamic compilation systems usually implement a strategy that leans towards under-specialization because performing method specialization may require a significant amount of compilation time, and thus should only be done when it greatly benefits execution speed.

- The amount of code growth considered acceptable by the compiler. Some method specialization strategies allow an unlimited amount of code growth, while other strategies only permit a set limit either as a specific number of specialized method allowed, or a specific code size limit measured in bytes.

- The estimated benefit of performing method specialization. Estimating the amount of speedup is not an exact science and estimation strategies are also dependent upon the compiler implementation. Estimation may take into consideration the following aspects that are examined in Section 8.3.2.

- The estimated cost of performing method specialization. After specialization an additional method needs to be compiled and optimized. Current compilers usually have some sort of cost estimation built-inthat may be utilized for method specialization decisions.

Existing method specialization techniques will be examined in following chapters.

## 4.6 Advantages and Disadvantages

Method specialization offers several advantages and disadvantages. The additional methods that must be compiled may contribute to an increase in compilation time. While longer compilation time is not an issue of concern in static compilation, dynamic-compilation systems perform optimization at run-time. Thus minimizing the cost of compilation is important for execution speed, making method specialization strategy selection important.

Additionally, a program that contains specialized methods can be significantly larger that the same program without specialized methods. This again depends upon the method specialization strategy implemented by the compiler.

The main advantage of method specialization is that it creates opportunities for other optimizations. In particular, call site devirtualization may create new opportunities for method inlining.

A call site devirtualization may eliminate the need for a method lookup. However, on modern compiler implementations, this does not result in significant speedup because most compilers implement efficient method lookup systems. Most of the speedup comes from the method inlining performed after a call site has be devirtualized. Method specialization — in concert with call site devirtualization, method inlining, and other optimizations — can yield execution speedups as high as 275% [26, 48].

Method specialization also allows programmers to write generic programs without loss of efficiency due to abstraction layers. Method specialization can remove the needed abstraction while allowing the programmer to generate programs in a way that is conducive to good program design practices.

Long-running programs are most likely to benefit from method specialization because specialized methods are more likely to be executed resulting in greater speedup. However, because specialized methods are generally created based on either online or offline profiling information, method specialization can suffer when the program context changes. If the profiling information becomes stale, specialized methods may no longer be in the execution path of the program and therefore may not utilized as often as predicted.

Method specialization may cause call sites that have only one target method, and are thus static, to become virtual call sites. In the example of Figure 4.3,

a new method *B.func()* is added to the class hierarchy. While this new method devirtualized the call site to method *foo()*, any method that calls method *func()* now has 2 methods to choose from. Thus a static call site has now become a virtual call site that requires method lookup and dispatch.

In programming languages like Java, all method calls to methods that are not defined as static are defined to be virtual, even if there is only one possible target. Therefore, in the example above there is no re-virtualization unless the compiler had first changed the virtual call site to a static call site.

## 4.7 The Focus of This Thesis

This thesis focus on method specialization with regards to virtual dispatching, because call site devirtualization may create opportunities for inlining which in turn may yield significant execution speedups.

The following chapters contain descriptions of previous method specialization techniques. These techniques will be discussed and compared along with a discussion of the suitability of using the framework within the Java environment.

# Chapter 5

# Customization

Customization was initially introduced by Craig Chambers and was created in response to the high cost of virtual method calls in object-oriented programming languages [26, 27, 28, 31, 33, 86]. Customization was created to help reduce the number of virtual method calls. The main motivations for customization were the large number of virtual call sites in object-oriented programs and the high cost, at the time, of dynamic dispatching. Customization reduces the overhead of performing method lookup at a virtual call site by replacing a polymorphic method call with a direct method call.

## 5.1 Description

Customization was originally created for the SELF programming language which was designed to be a pure object-oriented programming language. SELF does not allow for dynamic class loading.

Customization is a method specialization technique that creates a specialized version of each method for every class inheriting the method. As a result, the exact class of the receiver of a customized version of a method is known at compilation time. Thus, the compiler can devirtualize method call sites where the receiver object is "self". Because call sites with receiver objects to "self" are quite common in object-oriented programming languages, the speedups created from customization in SELF are significant.

## 5.2 Strategy

Customization uses a brute-force method as its specialization strategy. No concern is given for code growth or for the creation of duplicate methods when only one

would suffice. Customization sees that each subclass gets it's own copy of each method implementation. This strategy is the easiest to implement and does not require computation for specialization decisions.

## 5.3 An Example of Customization

Using the example from the previous chapter which is shown again in Figure 5.1, Figures 5.2, 5.3, 5.4, 5.5, 5.6 and 5.7 show what the classes would look like after customization (but without any additional optimizations).

```
public class A {
    A localVar;
    void func() {
        foo();
    }
    void func2(A param) {
        param.foo();
    }
    void func3() {
        localVar.foo();
    }
    void foo() { ... }
}
public class B extends A {
    void foo() { ... } // Overridden from class A
}
public class C extends B { ... }
public class D extends C {
    void foo() { ... } // Overridden from class B
}
public class E extends C { ... }
public class F extends A { ... }
```

Figure 5.1: A Sample Pseudo Java Class Hierarchy Before Method Specialization

In this example the methods have been duplicated from the superclass implementation and added into the class hierarchy. As a result, method *func()* now has only one possible target method for the call site *foo()* regardless of the receiver object type. This has effectively devirtualized this call site.

## 5.4 Results And Issues

Customization turned out to be an effective technique used in the SELF programming language and resulted in speedups of 1.5 to 5 times over executions without customization [28]. However these benchmarks were done on a small set of pro-

```
public class A {
    A localVar;
    void func() {
        foo();
    }
    void func2(A param) {
        param.foo();
    }
    void func3() {
        localVar.foo();
    }
    void foo() { ... }
}
```

Figure 5.2: Class *A* After Customization

```
public class B extends A {
    // Duplicated from A.func()
    void func() {
        foo();
    }
    // Duplicated from A.func2(A param)
    void func2(A param) {
        param.foo();
    }
    // Duplicated from A.func3()
    void func3() {
        localVar.foo();
    }
    void foo() { ... } // Overridden from class A
}
```

Figure 5.3: Class *B* After Customization

```
public class C extends B {
    // Duplicated from A.func()
    void func() {
        foo();
    }
    // Duplicated from A.func2(A param)
    void func2(A param) {
        param.foo();
    }
    // Duplicated from A.func3()
    void func3() {
        localVar.foo();
    }
    // Duplicated from B.foo()
    void foo() { ... }
}
```

Figure 5.4: Class *C* After Customization

```
public class D extends C {
    // Duplicated from A.func()
    void func() {
        foo();
    }
    // Duplicated from A.func2(A param)
    void func2(A param) {
        param.foo();
    }
    // Duplicated from A.func3()
    void func3() {
        localVar.foo();
    }
    void foo() { ... } // Overridden from class B
}
```

Figure 5.5: Class *D* After Customization

```
public class E extends C {
    // Duplicated from A.func()
    void func() {
        foo();
    }
    // Duplicated from A.func2(A param)
    void func2(A param) {
        param.foo();
    }
    // Duplicated from A.func3()
    void func3() {
        localVar.foo();
    }
    // Duplicated from B.foo()
    void foo() { ... }
}
```

Figure 5.6: Class *E* After Customization

```
public class F extends A {
    // Duplicated from A.func()
    void func() {
        foo();
    }
    // Duplicated from A.func2(A param)
    void func2(A param) {
        param.foo();
    }
    // Duplicated from A.func3()
    void func3() {
        localVar.foo();
    }
    // Duplicated from A.foo()
    void foo() { ... }
}
```

Figure 5.7: Class *F* After Customization

grams [32], and the execution of the SELF code was 2 to 3 times slower than their optimized C counterparts. Additionally the benchmarks used in the first evaluation of customization were fairly small. Later, when customization was tested on larger programs, the execution time gains were not as impressive and compilation was slow.

Customization resulted in a substantial increase in code size — by as much as a factor of three for some applications. Moreover, the larger the application, the larger the amount of customization performed [53]. This is a consequence of the brute-force technique of creating a specialized method for *every* applicable class that inherits the method. The examples in Figures 5.1 and Figures 5.2 through 5.6, illustrate that the space requirements for customization in a simple program is large. This requirement increases with the complexity of the class hierarchy.

Simple customization also suffers from both over-specialization and under-specialization. Over-specialization is a problem because methods are specialized without any consideration to costs and benefits. In many cases, several specialized methods are virtually identical and thus could be combined and used as a single method without any significant impact on program execution. Combining similar methods would result in a significant reduction in code increase.

In dynamic systems, customization can be done lazily by delaying the creation of specialized methods until it is actually needed (if at all). This strategy eliminates generating and compiling dead code for classes that do not need a customized

method, but does not eliminate the problem of over-specialization because a method that is invoked only once for a particular receiver class will generate a customized method. Additional problems with customization and dynamic compilation will be examined in the next chapter.

In addition to over-specialization, customization suffers from under-specialization which is caused by only specializing a method on the type of the receiver object. It may also be beneficial to specialize on classes of arguments to the method, or other properties of a method as mentioned before (See Section 4.3), but customization does not perform these actions.

## 5.5 Suitability For Java

Simple customization is not very suited for a dynamic compilation system. Because simple customization increases the number of methods to be compiled, and correspondingly the size of the program, the amount of time required to compile these methods increases proportionately. As compilation is done at runtime, spending execution time compiling methods that are either infrequently or never executed reduces execution speed for no particular reason. Therefore a dynamic compiler should use a cost/benefit analysis in order to reduce the number of new specialized methods.

Customization does not take dynamic class loading into consideration. While at the time, dynamic class loading was not a major concern, the broad usage of Java have made it more important today. Theoretically a dynamic compiler could insert new specialized classes into a dynamically loaded class when the class is loaded. However this technique still suffers from the code size growth problems discussed previously.

# Chapter 6

# Selective Specialization

*Profile-Guided Selective Method Specialization* [29, 48], also known as *Selective Specialization*, was created by Jeffrey Dean, Craig Chambers and David Grove as an extension to customization. Selective specialization was specifically created to address the problems of over-specialization and under-specialization that customization did not address.

## 6.1 Description

Selective specialization performs method specialization by selectively choosing opportunities where the benefits are estimated to be large. This selection helps reduce over-specialization by limiting the number of methods specialized to only opportunities that are likely to produce execution speedup. Class hierarchy analysis is also preformed to determine if multiple classes could share a single specialized method, as opposed to customization which requires a single distinct method for each class. Selective specialization also takes on the under-specialization problem by specializing not only on just the receiver object of call sites, but also on any formal argument to a method.

Selective specialization begins by collecting profiling information and creating a weighted dynamic call graph (see Section 2) to identify frequently executed methods and call sites. The algorithm employed by selective specialization focuses on *specializable call sites*, which are defined as a dynamically dispatched pass-through call site (see Section 2).

Once highly executed specializable call sites have been found, class hierarchy analysis is performed. This analysis finds the largest subset of classes that contain methods that the specialized method could be bound to. If a subset exists, then

the specialized method is created. This specialization algorithm allows a single specialized method to be used instead of multiple similar specialized methods.

When call sites are specialized only on the type of the receiver object, once a specialized method has been created, existing mechanisms, such as virtual method tables, can be used to select the appropriate version for each method site. However if method specialization is done on the argument to a method, support for multi-methods is required by the runtime system.

## 6.2 Strategy

In order to determine which call sites are considered to be frequently executed, the strategy implemented by the authors is a simple heuristic where if the weight is above a user-defined threshold, then the call site is marked for specialization. The specific threshold number used by Chambers, Dean and Grove was 1000 invocations [29]. However they list several deficiencies with this approach.

- The heuristic does not take into consideration the code growth caused by the creation of the specialized method.
- The heuristic consider neither the total size of the specialized method, nor the number of specialized methods needed to statically bind a call site.
- The heuristic does not take into consideration that certain call sites allow for greater execution speedup. Because of the difference in effects of certain optimizations (such as inlining) that can be performed after specialization, the resulting execution speedup can vary significantly. The authors suggest that it may be possible to estimate the benefits of optimizations after specialization and then use that in the heuristic to determine if the specialized method is beneficial.

In spite of these shortcomings, they argue that a better heuristic is not necessary because of the excellent speedup they gained using this simple heuristic.

## 6.3 Example

Consider the class hierarchy information shown in Figure 6.1 that is based on the example from previous chapters. Although selective specialization is used with pro-

gramming languages that do not provide dynamic class loading, the examples in this chapter are presented in pseudo Java for consistency.

```
public class A {
    A localVar;
    void func() {
        foo();
    }
    void func2(A param) {
        param.foo();
    }
    void func3() {
        localVar.foo();
    }
    void foo() { ... }
}
public class B extends A {
    void foo() { ... }
}
public class C extends B { ... }
public class D extends C {
    void foo() { ... }
}
public class E extends C { ... }
public class F extends A { ... }
```

Figure 6.1: A Sample Pseudo Java Class Hierarchy

Profile-guided method specialization requires a weighted program call graph constructed from profile data that describes:

- Each call site in the program;
- The set of methods invoked at each call site;
- The corresponding number of executions of each call site.

A call site may have multiple callee's because the receiver object may change due to dynamic dispatching. Figure 6.2 shows a partial weighted program call graph for a given execution of the call site *foo()* in method *A.func()* from Figure 6.1.

The selective specialization algorithm visits each node in the call graph looking for dynamically-dispatched call sites that are executed over 1000 times. Given the call graph from Figure 6.2, the call site *foo()* in *A.func()* has a target of *A::foo()* 500 times, *B::foo()* 1500 times, and *D::foo()* 0 times. As *B::foo()* has an invocation count over 1000, a specialized method with a direct call to *B.foo()* is marked for creation.

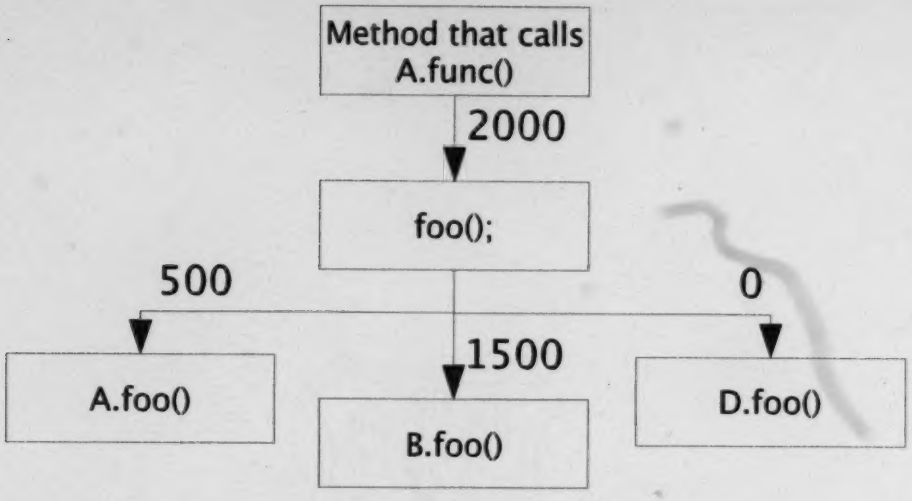


Figure 6.2: A Sample Weighted Call Graph For the Class Hierarchy in Figure 6.1

The algorithm then computes the subset of classes that would allow static binding of the call site, and then creates a corresponding specialized method. This is performed by examining the class hierarchy and the target method's reachability. The method reachability for the *foo()* method is shown in Figure 6.3.

Figure 6.3 shows the program class hierarchy divided into regions that show the reachability of method *foo()*, *i.e.* which version of method *foo()* is executed based on the receiver object type. The example shows that method *foo()* is defined in classes *A*, *B*, and *D*. Thus a *foo()* method call to an object of runtime type *B*, *C*, or *E* would result in the execution of the definition given in class *B*. A call from an object of runtime type *D* would result in the definition in class *D* being executed and a call from an object of type *A* or *F* would result in the execution of the definition given in class *A*.

The method reachability of *foo()* indicates that it is possible to make the call site *foo();* in method *A::func()* static for classes *B*, *C*, and *E* when creating a specialized method. This specialized method would be called whenever the method's receiver object is of type *B*, *C* or *E*.

Figure 6.4 shows the virtual function tables for the class hierarchy before method specialization. Figure 6.5 shows the changes to the virtual function tables after the specialized method is added into the class hierarchy. When the specialized method is inserted into the class hierarchy, the classes *B*, *C* and *E* call the specialized method

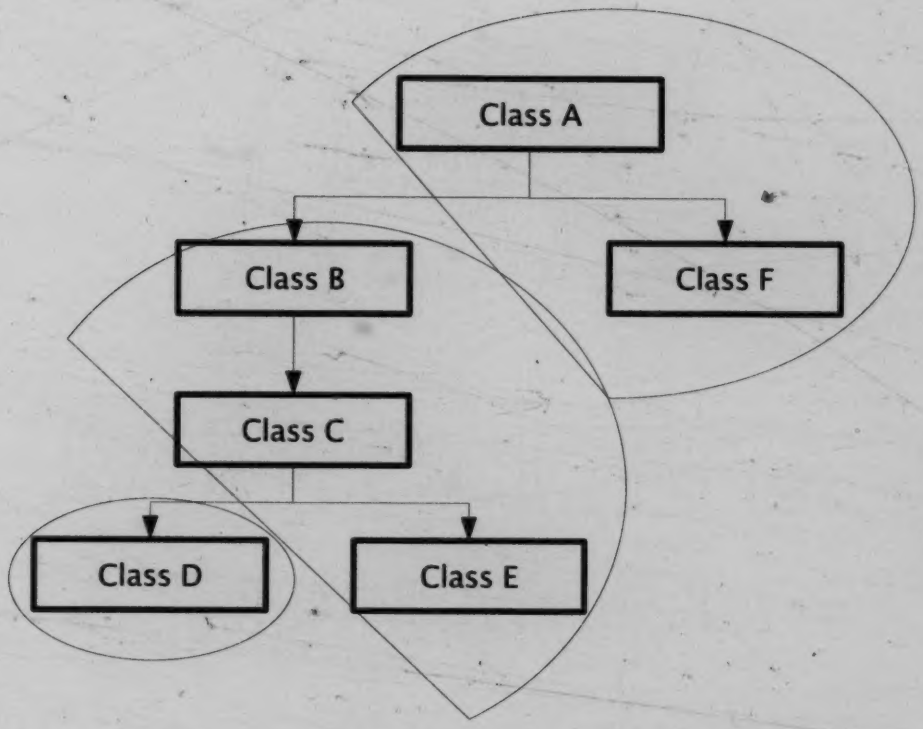


Figure 6.3: Method Reachability of method *foo()*

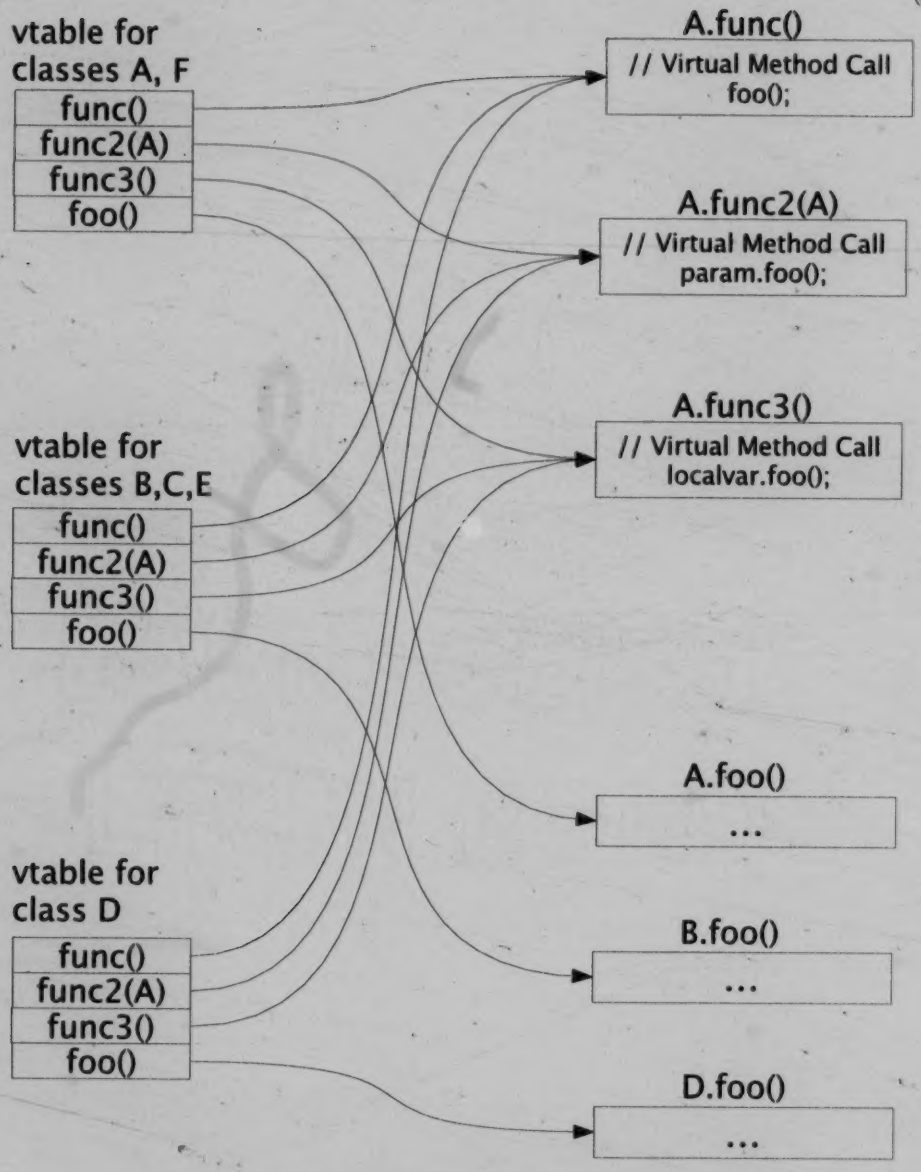


Figure 6.4: Virtual Function Tables Before Method Specialization

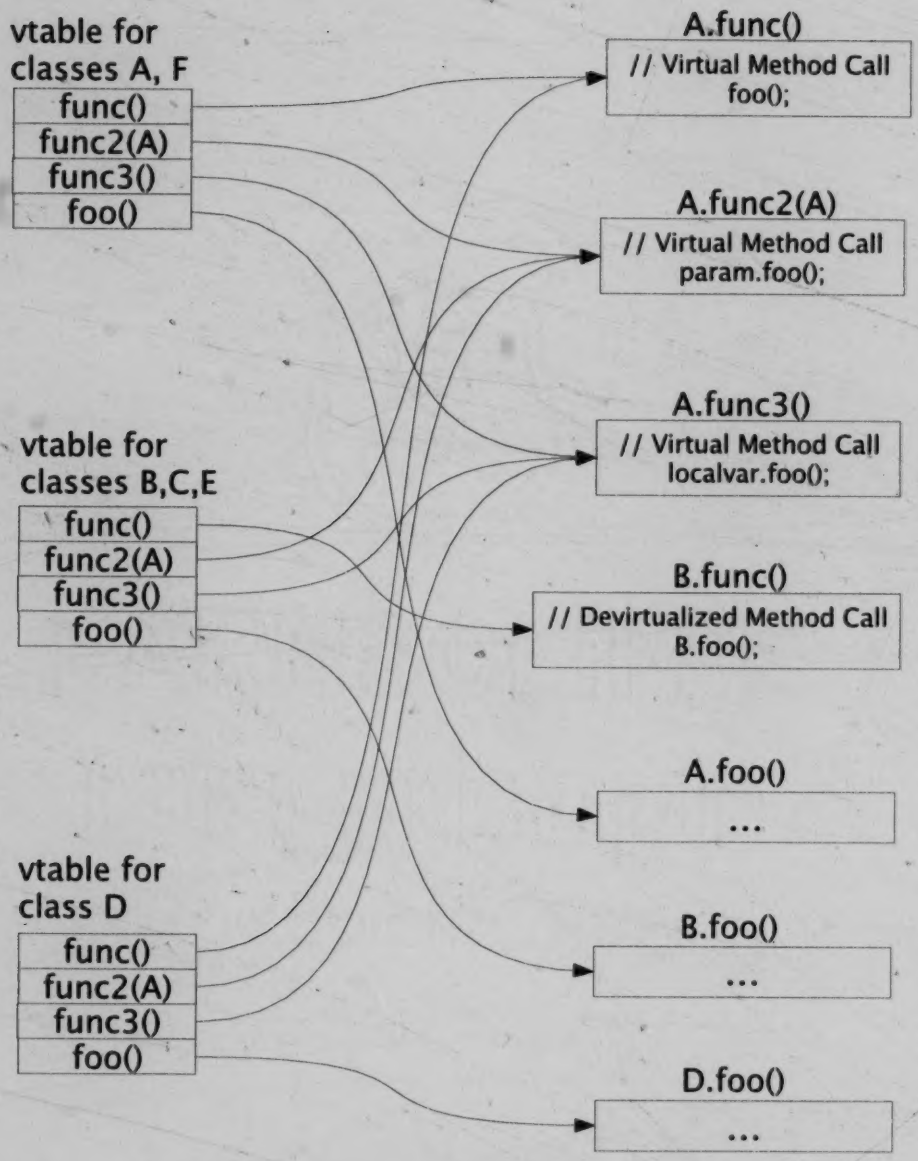


Figure 6.5: Virtual Function Tables After Method Specialization

*func()* while the other classes call the original method *func()*.

## 6.4 Cascading Specializations

The concept of cascading specializations was introduced to deal with the side-effect of turning static call sites into virtual call sites as a result of method specialization. Figure 6.6 shows the previous example after method specialization, but with an added method, *callsFunc()*, that has a call site to method *func()*.

```
public class A {
    A localVar;
    // This is the original method
    void func() {
        foo();
    }
    // This is the new specialized method
    void func() {
        B.foo(); // Devirtualized method call site
    }
    void func2(A param) {
        param.foo();
    }
    void func3() {
        localVar.foo();
    }
    // New method that calls method func()
    void callsFunc() {
        // Before specialization this was a static call site.
        // Now it is a virtual call site
        func();
    }
    void foo() { ... }
}
public class B extends A {
    void foo() { ... }
}
public class C extends B { ... }
public class D extends C {
    void foo() { ... }
}
public class E extends C { ... }
public class F extends A { ... }
```

Figure 6.6: A Sample Pseudo Java Class Hierarchy Showing How Call Sites Can Become Virtual After Specialization

In this example the call site *func()* in method *A::callsFunc()* was static before specialization because there was only one target method. However, after specialization there are two potential target methods thus making the call site virtual.

Cascading specializations attempts to eliminate this side-effect by specializing the callers of the original method. The technique for specializing the caller is the same as before: the call site execution count is taken into consideration and must be above the defined threshold of 1000 and the call site must be a specializable call site. In the example, specializing method *A.callsFunc()* could cause static call sites to this method to become virtual. Thus the algorithm continues up the hierarchy until either no specializable call sites are found, or the specializable call site is not executed frequently enough, *i.e.* the site is executed less than 1000 times. Figure 6.7 shows the class hierarchy after cascading method specialization has been done on method *A.callsFunc()*.

```
public class A {
    A localVar;
    // This is the original method
    void func() {
        foo();
    }
    // This is the new specialized method
    void func() {
        B.foo(); // Devirtualized method call site
    }
    void func2(A param) {
        param.foo();
    }
    void func3() {
        localVar.foo();
    }
    // Original method with virtual call site
    void callsFunc() {
        func();
    }
    // New cascaded specialized method
    void callsFunc() {
        A.func(); // Devirtualized call site
    }
    void foo() { ... }
}
public class B extends A {
    void foo() { ... }
}
public class C extends B { ... }
public class D extends C {
    void foo() { ... }
}
public class E extends C { ... }
public class F extends A { ... }
```

Figure 6.7: A Sample Pseudo Java Class Hierarchy After Performing Cascading Method Specialization

## 6.5 Results

Working with four Cecil benchmarks, Dean et al. recorded performance speedups of between 65 to 275%, significantly better than class hierarchy analysis at 24 to 70%, and customization at 26 to 125% [48]. Selective specialization was able to eliminate 54-66% of all virtual call sites, better than customization which was able to eliminate 35-61% of all virtual call sites. Selective specialization was able to eliminate more virtual call sites over customization as it focuses on specializable call sites as opposed to only focusing on call sites with receiver targets of the current object. Additionally, the code growth due to selective specialization was only 4-10% for those benchmarks while achieving the highest runtime speedup.

## 6.6 Suitability For Java

The authors argue that selective specialization is a practical optimization for runtime compilers where recompilation could be triggered when a method call site reaches the 1000 invocation threshold. While this is true, the strategy employed by selective specialization does not consider phase shifts that can take place in long running programs, and dynamic class-loading requires recompilation for broken specialized methods.

# Chapter 7

# Automatic Program Specialization

The notion of Automatic Program Specialization [11, 109, 110, 111] was introduced to add method specialization support to the Java programming language. Automatic program specialization differs from the previous two method specialization techniques because it requires a language extension to the programming language as opposed to being a compiler code transformation.

## 7.1 Description

The goal of automatic program specialization is to specialize Java programs to specific values as these values become available. Because automatic specialization may happen both at compile time and at run time, their framework requires a language extension to declare what values to specialize the program to. Automatic program specialization uses a declarative approach presented by Volanschi *et al.* [126] called *Specialization Classes*. This approach gives a high-level language the ability to specify on what basis a program is to be specialized, including:

- *Program Components*: Can specify which part of the code to specialize (the whole program, a set of methods, a single method, or a block of code).
- *Specialization Context*: Can specify values of variables to specialize the program to.
- *Incremental Specialization*: When specializing to a set of values, the predicates may not become true all at the same time. Thus, declarative specialization allows specialization with regard to a sequence of specialization contexts.

These new specialization classes along with the original Java program classes are input for JSpec [109], an off-line automatic program specializer. JSpec then can output one of three program types:

- *Java Source Code*: The specialized Java source code is represented as an AspectJ aspect.
- *C Source Code*: The specialized portion can be output as C source code that is executed in the Harissa [98] environment. Harissa is a Java to C compiler which is used by Automatic Program Specialization to translate a program from Java into C.
- *Binary Code*: Output can also be binary code for execution in the Harissa environment.

The process used by JSpec involves taking the Java program classes and specialization classes and translating them to C using Harissa. These C files are input for Tempo [42], a specializer for C programs, which creates a C program that is specialized based on the specialization classes. The C code is then translated back into Java with the specialized code represented as an AspectJ aspect. AspectJ is then used to weave the specialized code and original code together.

Having the specialized code encapsulated in an AspectJ aspect allows complete separation of the original program and the specialized code. This also gives the ability to plug-in and un-plug the specialized code simply by selecting whether the aspect should be included or not.

Method selection between the specialized method and the original method is handled by a pointcut. The pointcut executes a check to determine if the object's state is the same as specified in the specialization context. If the state is the same, then the specialized method is executed otherwise the original method is executed instead. This check is performed before each invocation of the method.

Dynamic class loading is implicitly handled because JSpec only takes a subset of the program and specializes it. However if a class can be dynamically loaded in the program slice, it must be included in the slice in order to be specialized.

## 7.2 Strategy

The strategy used is completely dependent on the programmer. The programmer chooses where specialization should be performed and on what data. Thus the strategy rests on how aggressively the programmer would like the compiler to specialize the program.

## 7.3 Example

As automatic program specialization requires specialization input from the programmer, the previous example needs to be extended to include specialization classes that indicate what needs to be specialized. Figure 7.1 includes the original program with the specialization class of Figure 7.2.

```
public class A {
    A localVar;
    void func() {
        foo();
    }
    void func2(A param) {
        param.foo();
    }
    void func3() {
        localVar.foo();
    }
    void foo() { ... }
}
public class B extends A {
    void foo() { ... }
}
public class C extends B { ... }
public class D extends C {
    void foo() { ... }
}
public class E extends C { ... }
public class F extends A { ... }
```

Figure 7.1: A Sample Pseudo Java Class Hierarchy

```
specclass specA specializes A {
    D localVar;
    void func3();
```

Figure 7.2: A Sample Specialization Class

In Figure 7.2 the specialization class *specA* contains one specialization instance.

The specialization class states that method *func3()* should be specialized with regards to the information that variable *localVar* is of type *D*. Using this information the call site *localVar.foo()* can be devirtualized because there is only one target method when the receiver object is of type *D* . The resulting call site would be a static call to *D.foo()* .

```
aspect SpecializeFunc3 {
    // Specialized Method
    private void A.newSpecFunc3() {
        D.foo(); // Devirtualized call site
    }

    pointcut func3Called() :
        call(void func3())
        && target(receiverObject);

    void around(Object receiverObject) :
    func3Called(receiverObject) {
        if(guard()) { // Check specialization context
            // Call specialized method
            return A.newSpecFunc3();
        } else {
            // Call original method
            return localVar.specFunc3();
        }
    }

    private boolean guard() {
        if(localVar instanceOf D) {
            return true;
        } else {
            return false;
        }
    }
}
```

Figure 7.3: The Corresponding Aspect Containing Specialized Code

In Figure 7.3, the method *A.newSpecFunc3()* specifies the new specialized method. The pointcut *func3Called()* specifies the instance during program execution when method *func3()* has been called. Next appears the advice *around(receiverObject)* which is called whenever the program reaches a *func3Called()* pointcut (whenever method *func3()* is called). Method selection then takes place by calling the private method *guard()* to check if the current program state matches the specialization context to decide whether the specialized method or the original method should be called.

## 7.4 Results

The authors used various generic benchmark programs to generate their data. Automatic program specialization saw code increases of 2.8 times on average, with one benchmark resulting in a 40 times size increase. Speed increases averaged 2.4 times over unspecialized code on the IA32 architecture (3.0 times speedup for SPARC). On the benchmarks tested for the IA32 architecture, most benchmarks benefited a bit from specialization, while a select few benefited greatly (12 times speedup). The recorded speedups do not include any benefits from inlining. The authors found that the benefits from inlining varied greatly depending on the JIT used.

## 7.5 Suitability For Java

Automatic program specialization requires that all affected classes be loaded before the specialization takes place. This restriction seriously affects the usefulness of automatic specialization for Java. Dynamic class loading is an important feature of Java that allows the user to load information, such as a device driver, at run time without requiring that the class be available at compile time.

Method specialization can benefit from run-time profiling information and make specialization decisions about receiver-types that are often passed at call sites.

Automatic program specialization gives more control to the programmer by allowing control over where and what to specialize. This control comes at the expense of a more complex program and a more complex programming language. Additionally the programmer may not know what values will stay consistent through the program execution and thus benefit from method specialization. This is also evident with specialization on values obtained from user interfaces because the programmer would have to predict what values the majority of users would use.

Without recompilation, automatic program specialization offers no adaptability across different input sets. While a program could be specialized to a specific data set to take full advantage of the specialized code, an input data set inconsistent with the data set used when compiled would completely forfeit the speedups. A dynamic compiler, however, could respond at run-time and provide speedup regardless of the data set used.

# Chapter 8

# A New Method Specialization Framework For Java

## 8.1 Why A New Framework?

While the previous frameworks reviewed offer significant speedups, none of them are ideally suited for a Java compiler optimization framework. Customization would require far too much compile time and space, selective specialization requires recompilation for changes that happen to the class hierarchy as a result of dynamic class loading, and automatic program specialization is an off-line optimization that does not explicitly handle dynamic class loading or consider on-line profiling information.

Thus a new framework that considers all this information is needed. In this chapter I propose a new method specialization framework that deals with the issues when performing method specialization with Java.

## 8.2 Method Specialization And Java Issues

We first highlight the issues that method specialization implementations face when targeting Java, minus the redundant issues already addressed by previous research.

### 8.2.1 The Java Super Keyword

The *super* keyword in the Java programming language offers some unique constraints for method specialization. Consider the program in Figure 8.1.

Examining the class hierarchy, it may make sense, based on profiling information, to create a new specialized method *A.func()* that contains a devirtualized call site to *A.foo();* (we ignore the issue of dynamic class loading for simplicity). Figure 8.2 shows what the program would look like after performing method specialization.

```
public class A {
    void func() {
        foo();
    }
    void foo() { ... }
}

public class B extends A {
    void foo() { ... } // Overridden method from Class A
    void something() {
        super.func();
    }
}
```

Figure 8.1: A Sample Pseudo-Java Program Using the **super** Keyword

```
public class A {
    // New specialized method
    void specFunc() {
        A.foo(); // Devirtualized Call Site
    }
    // Original method
    void func() {
        foo(); // Virtual call site
    }
    void foo() { ... }
}

public class B extends A {
    void foo() { ... } // Overridden method from Class A
    void something() {
        super.func();
    }
}
```

Figure 8.2: A Sample Pseudo-Java Program Using the Super Keyword After Method Specialization

When method *B.something()* executes call site *super.func()*, it may be tempting to execute the specialized method *A.specFunc()*. However, Section 15.11.2 of the Java Language Specification [64] states that when the target of a call site is a superclass — as dictated by the *super* keyword — the instructions executed are taken from the corresponding method in the superclass. But any virtual-method call sites in those instructions must be relative to the caller class, not to the superclass. Therefore when the call site *super.func();* is executed, method *A.func()* is the target method but the call site *foo();* is relative to class *B*, the caller class, and thus the target method must be method *B.foo()* and not *A.foo()*. This behavior needs to be addressed when performing method specialization in Java to ensure correct execution.

To handle this constraint, in this framework whenever a message is sent to the "super" object, we never dispatch to a specialized method. Instead we send a message to the original method which contains virtual call sites. This mechanism ensures correct execution.

### 8.2.2 Dynamic Class Loading

Dynamic class-loading is one of the main features of the Java programming language. However the presence of dynamic class-loading can affect the opportunities for compiler optimizations such as method specialization. Figure 8.3 shows an example.

```
public class A {
    void func() {
        foo();
    }
    void foo() { ... }
}

public class B extends A {
    void foo() { ... } // Overridden method from Class A
}

public class C extends B { ... }
```

Figure 8.3: A Sample Pseudo-Java Program

Figure 8.3 shows a pseudo-Java program that, without dynamic class-loading, would be a candidate for method specialization. Method *func()*, implemented in class *A*, contains a virtual call site to method *foo()*. In the class hierarchy shown in

Figure 8.3, method *foo()* contains overriding declarations in class *B*, *i.e.*, if method *func()* is called by an object of runtime type *B* or a subclass, method *B.foo()* must be called. Thus we can create a specialized method *B.func()* that would contain a devirtualized call site to method *B.foo()* as shown in Figure 8.4.

```
public class A {
    void func() {
        foo();
    }
    void foo() { ... }
}

public class B extends A {
    // New specialized method
    void func() {
        B.foo(); // Devirtualized Call Site
    }
    void foo() { ... } // Overridden method from Class A
}

public class C extends B { ... }
```

Figure 8.4: A Sample Pseudo-Java Program After Method Specialization

This newly inserted specialized method contains devirtualized call sites that eliminate the overhead of performing a method call. While execution would be correct for the current class hierarchy, dynamic class loading allows a new class to be loaded at any point in the program execution. Figure 8.5 shows the Java program after a new class has been dynamically loaded, and after method specialization.

The newly loaded class *D* breaks the method specialization. Now, if method *func()* is called from an object of runtime type *D*, then method *B.func()* will be executed which, in turn, will incorrectly execute methods *B.foo()* due to the devirtualized call sites.

The new framework explicitly handles dynamic class loading by adding additional logic to the class loader of the Java Virtual Machine. In the new framework the virtual method table of dynamically loaded classes is built in such a way that specialized methods can be called where execution would be correct, or else a call to the original method through a virtual call takes place.

```
public class A {
    void func() {
        foo();
    }
    void foo() { ... }
}

public class B extends A {
    // New specialized method
    void func() {
        B.foo(); // Devirtualized Call Site
    }

    void foo() { ... } // Overridden method from Class A
}

public class C extends B { ... }

public class D extends C {
    void foo() { ... } // New overridden method from Class B
}
```

Figure 8.5: A Sample Pseudo-Java Program After Dynamic Class Loading

## 8.3 Framework Description

This section describes the new method specialization framework assuming its implementation in the Jikes Research Virtual Machine. The description is illustrated with the same example used in previous chapters. For convenience, the example is presented again in Figure 8.6, with one small modification. In order to illustrate how the framework handles dynamic class loading, now classes $D$ and $E$ are dynamically loaded after method specialization has been performed.

### 8.3.1 Profiling Collection

Information first needs to be gathered with regards to which call sites are considered to be hot, *i.e.*, frequently executed, and what the receiver-type class distributions are for those call sites. The best instrumentation method for collecting this information is the technique developed by Arnold and Ryder [19], which allows for light-weight profiling collection.

Upon finding a frequently executed virtual call site, the compiler uses a heuristic to determine whether method specialization should be performed to devirtualize the call site.

For instance, in the Jikes Research Virtual Machine optimization subsystem, if a

```
public class A {
    A localVar;
    void func() {
        foo();
    }
    void func2(A param) {
        param.foo();
    }
    void func3() {
        localVar.foo();
    }
    void foo() { ... }
}
public class B extends A {
    void foo() { ... }
}
public class C extends B { ... }
public class F extends A { ... }
```

Figure 8.6: A Sample Pseudo-Java Class Hierarchy

```
public class D extends C {
    void foo() { ... }
}
public class E extends C { ... }
```

Figure 8.7: Dynamically-Loaded Classes Loaded Into the Class Hierarchy in Figure 8.6 After Performing Method Specialization

method is hot enough to trigger recompilation at level 2, then the method is checked for specialization opportunities.

Assume, for example, that method *A.func()* is flagged for recompilation at level 2 which includes the method specialization optimization. The compiler then looks at the run-time profiling information for receiver object-types at all the call sites in the method. A sample of receiver-type profiling information for the *foo()* call site is given in Figure 8.8.

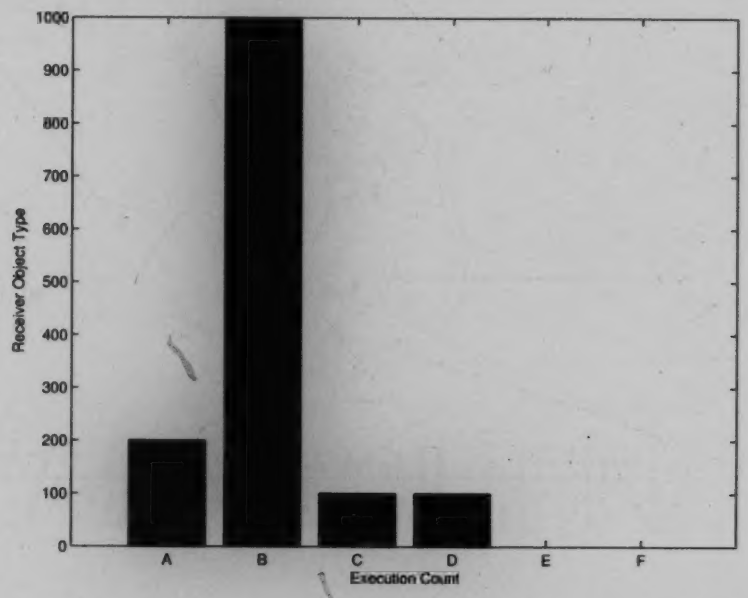


Figure 8.8: Sample Receiver-Type Profiling Information for Call-Site foo()

The information from Figure 8.8, is applied to the applies-to sets for the target method (method *foo()* in this case). The applies-to sets for method *foo()* can be found in Figure 6.3.

The receiver-object-type profiling information for each applies-to set reveals that the set { *A*, *F* } has received a total of 200 message sends to method *foo()*; set { *B*, *C* , *E* } has received a total of 1200 message sends; and set { *D* } has received no message sends. The receiver class distribution is highly peeked towards the { *B*, *C* , *E* } class set. The exact percentage of messages that a class set must receive to trigger method specialization is left up to the compiler heuristic strategy. In the example, having a class set that receives a high amount of method sends, along with

the ability to inline the *foo()* call site makes *A.func()* a good candidate for method specialization.

### 8.3.2 Aspects to Consider in Method Specialization Decisions

Modern compilers use profiling feedback information to estimate the cost/benefit ratio for a given code transformation. The strategy used in selective specialization is to require the programmer to specify a threshold above which the code transformation takes place [48]. This strategy is unsuitable for dynamic compilation environment because it ignores several important characteristics of these environments:

- *The frequency of the method targeted for specialization.* A method being considered for specialization must be executed frequently enough to justify the cost of performing the optimization. The higher the frequency of a method, the higher the likelihood that the method should be specialized.

- *Size of the method.* The decision must take into consideration the size of the method because this information is necessary to estimate how long it will take to compile and optimize the method. The method size also allows the compiler to calculate how much code growth is going to happen as a result of replicating the method.

- *Total cumulative size of specialized methods.* Keeping track of the total cumulative size of specialized methods allows tracking of code growth. The total amount of code growth allowed depends on the compiler strategy. Excessive code growth may cause excessive memory usage, instruction cache conflicts, and register spills due to increased variables and intermediate values [129].

  Code growth may also be kept in check through the removal of specialized methods based upon their usage within a specific time period. This option has not yet been researched and is identified as an area of future work described in Section 12.3.

- *The number of duplicate devirtualizable call sites.* The primary focus of method specialization is to devirtualize call sites, and therefore, open up opportunities for other optimizations. The higher the number of duplicate call sites that can

be devirtualized in a single specialized method, the more likely the method should be specialized.

- *The execution count of the devirtualizable call sites.* The execution count of the call sites that can be devirtualized should also be taken into consideration. Because control flow can cause instructions to be executed multiple times, a compiler strategy should also take into consideration the execution count of the devirtualizable call sites.

- *Potential for optimization.* The most important reason to perform method specialization is to create opportunities for other optimizations. Thus the compiler should attempt to make some estimation as to what optimizations will benefit from specialization, and how much benefit it will provide.

  The potential to inline the targeted call site should be weighted very heavily by the heuristic. Without inlining, merely devirtualizing a call site will likely not result in a large amount of speedup. Therefore, in most instances, method specialization should not be performed if the targeted call site cannot be inlined.

- *Receiver-Object-Type Distribution*: The receiver-object-type profiling information for the targeted call site must be compared with the applies-to set for the target method to determine the class distribution for each applies-to set. Using this information, the compiler must make a decision determining when an applies-to set receives enough message sends to trigger method specialization.

  The compiler may use either the absolute number of messages sent, or a percentage of the total message sends to decide if specialization must take place. Additionally the compiler could chose to apply a certain weight to the heuristic based on the frequency.

### 8.3.3 Optimization Transformation

Once a decision has been made to perform method specialization, the compiler takes the method source code and splits it into two. A pass is made through the newly duplicated source code to find the call sites that are flagged for devirtualization, where the compiler replaces the virtual call site with a direct call site. After devirtualization, the method can be passed on to the optimization framework where inlining and other optimizations are performed.

```
public class A {
    A localVar;
    void func() {
        foo();
    }
    void func2(A param) {
        param.foo();
    }
    void func3() {
        localVar.foo();
    }
    void foo() { ...}
}
public class B extends A {
    // New specialized method
    void func() {
        B.foo(); // Devirtualized call site
    }
    void foo() { ...}
}
public class C extends B { ...}
public class F extends A { ...}
```

Figure 8.9: A Sample Pseudo-Java Class Hierarchy After Method Specialization

Figure 8.9 shows a representation of what the class hierarchy would look like after method specialization. Figure 8.10 shows the corresponding virtual method tables (targets for methods *func2(A)* and *func3()* are left out for visibility). The class hierarchy now shows two *func()* methods. In order to select between the original method and the specialized method, the virtual method table needs to be able to correctly select the appropriate method, and the class loader requires additional logic to handle dynamic class loading.

### 8.3.4 Method Selection

To decide if the original method or the specialized method is to be used, a calculation needs to be done on the applies-to set for the target method of each call site being devirtualized.

**Specialized Class** A specialized class is defined as a class that has a virtual method table that targets at least one specialized method.

From the example in Figure 8.9, classes *B* and *C* would be specialized classes as they both target specialized method *func()*.

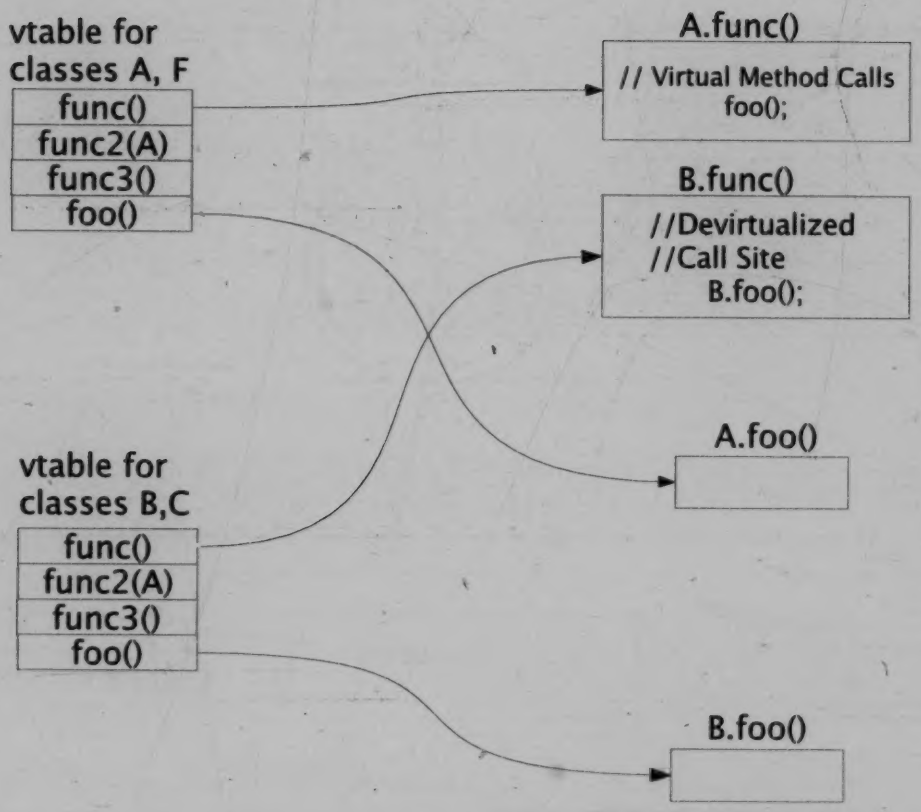


Figure 8.10: The Virtual Method Table After Method Specialization

**Specialized Target Methods Set** The set of methods that are the target of call sites that have been devirtualized in a specialized method.

From the example in Figure 8.9, the Specialized Target Methods Set would be { *B.foo()* } because that is the only target method of the call sites devirtualized in the specialized method *B.func()*.

**Specialized Method Applies-to Set** The applies-to sets for each method in the Specialized Target Methods Set are intersected together creating the specialized method applies-to set. The resulting set is the set of applicable classes that can call the specialized method with correct execution.

Again, in the example in Figure 8.9, calculating the specialized method applies-to set involves taking the applies-to set for *B.foo()*, as it is the target method for the devirtualized call site *foo()*. Thus the specialized method applies-to set would be { *B*, *C* }. Eventually when class *E* is loaded, it will be added to that set as it does not contain an overriding definition of method *foo()*.

Once this set has been calculated, the virtual method table needs to be changed to allow the specialized method to be executed in the correct instances. For each class in the specialized method applies-to set, the target method is changed to point to the specialized method instead of the original method. All other classes are left alone and point to the original method they pointed to before method specialization.

### 8.3.5 Class Loader Changes

In order to handle dynamic class loading, additional logic needs to be inserted into the class loader in order to maintain correct execution. When loading a new class, if it's superclass calls a specialized method, the method definitions in the newly loaded class need to be examined to see if they override any methods in the superclass's specialized target methods set. If a method has been overridden, then the new method must call the original method with the virtual call sites, not the specialized method, in order to maintain correct execution. Otherwise the specialized method can be called while maintaining correct execution.

In the example of Figure 8.11, when class *E* is loaded, the class loader identifies method *func()*, which is inherited by class *C*, as a method that has been specialized. This triggers a search of method definitions for class *E* to see if any method is overridden in the Specialized Target Methods Set for *func()*. In this instance, no

```
public class A {
    A localVar;
    void func() {
        foo();
    }
    void func2(A param) {
        param.foo();
    }
    void func3() {
        localVar.foo();
    }
    void foo() { ... }
}
public class B extends A {
    // New specialized method
    void func() {
        B.foo(); // Devirtualized call site
    }
    void foo() { ... }
}
public class C extends B { ... }
public class E extends C { ... }
public class F extends A { ... }
```

Figure 8.11: The Class Hierarchy After Dynamically Loading Class *E*

methods are overridden allowing class *E* to call the specialized version of method *func()*. Figure 8.12 shows the virtual method tables after the class *E* is loaded.

When class *D* is loaded as illustrated in Figure 8.13, the same class loader check shows that class *D* contains an overriding declaration of method *foo()*, which is contained in the Specialized Target Methods Set. In order to maintain correct execution, class *D's* virtual method table must point to the original method containing virtual method calls. Figure 8.14 shows the corresponding virtual method table.

## 8.4 Costs of Performing Method Specialization

As with any compiler optimization, there is an associated cost to method specialization. This cost is particularly important for programs that are executed with a virtual machine because optimizations are performed at run-time. Thus it is very important to keep the cost of optimizations as low as possible to prevent noticeable lags in program execution.

This section discusses the costs associated with implementing this method specialization framework. This has only been discussed theoretically as there is no implementation to test actual costs.

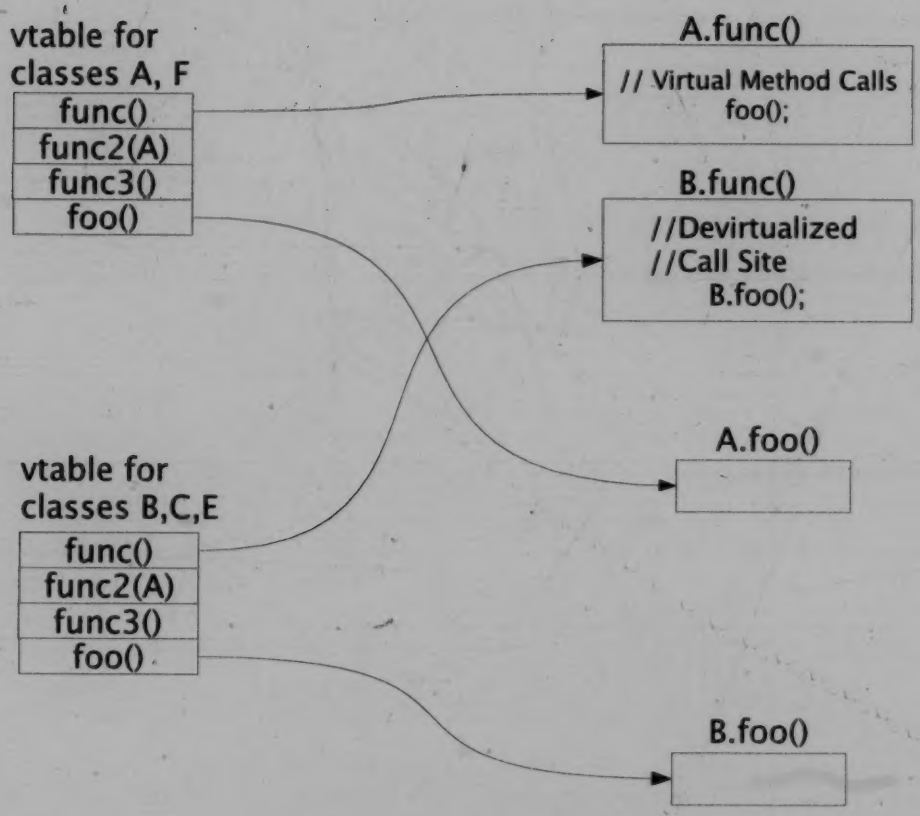


Figure 8.12: The Virtual Method Table After Dynamically Loading Class *E*

```
public class A {
    A localVar;
    void func() {
        foo();
    }
    void func2(A param) {
        param.foo();
    }
    void func3() {
        localVar.foo();
    }
    void foo() { ... }
}
public class B extends A {
    // New specialized method
    void func() {
        B.foo(); // Devirtualized call site
    }
    void foo() { ... }
}
public class C extends B { ... }
public class D extends C {
    void foo() { ... }
}
public class E extends C { ... }
public class F extends A { ... }
```

Figure 8.13: The Class Hierarchy After Loading Class *D*

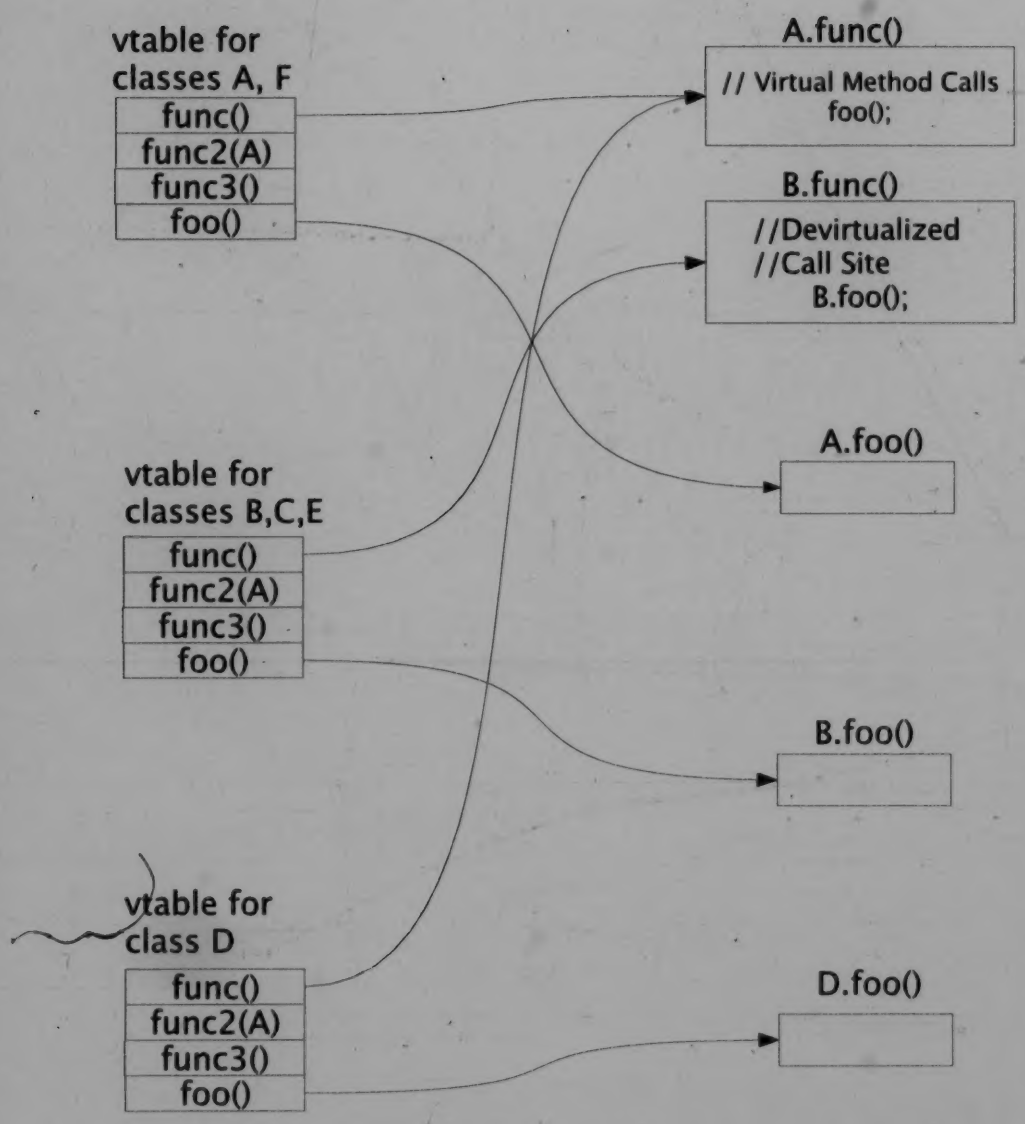


Figure 8.14: The Virtual Method Table After Dynamically Loading Class *D*

### 8.4.1 Profiling Information

The cost of performing instrumentation at run-time used to be quite heavy and would cause significant program slowdown. However the introduction of the instrumentation technique designed by Arnold and Ryder have reduced the cost of instrumentation to an overhead of 3 to 6% [19].

Instrumenting method frequency involves inserting a counter incrementing instruction at the start of a method. Receiver-type instrumentation is a bit more complex. Counter-increment instructions need to be inserted immediately before the targeted call sites. These counter instructions need to determine the type of the receiver object (the cost of this determination is compiler-dependent) and then increment the correct value in a 2-dimensional array indexed by the call site and the receiver-object type.

```
public class A {
    A localVar;
    void func() {
        foo();
    }
    void func2(A param) {
        param.foo();
    }
    void func3() {
        localVar.foo();
    }
    void foo() { ... }
}
```

Figure 8.15: Methods from Class *A* in Figure 8.6 to Show Receiver Object-Type Profiling Information

| Call Site / Class | A | B | C | D | E | F | Total |
|---|---|---|---|---|---|---|---|
| A.func()-foo()-Line 1 | 10 | 100 | 50 | 1000 | 40 | 0 | 1200 |
| A.func2(A)-param.foo()-Line 1 | 50 | 75 | 25 | 3000 | 0 | 0 | 3150 |
| A.func3()-localVar.foo()-Line 1 | 1000 | 1000 | 0 | 0 | 1 | 1 | 2002 |

Table 8.1: Example Receiver-type Profiling Information for Call Sites from Figure 8.15

Figure 8.15 shows a subset of the previous Java program. Table 8.1 shows sample receiver-type profiling information for each call site in Figure 8.15. For each call site, the run-time object-type of the receiver object is recorded in the appropriate column upon each execution.

### 8.4.2 Calculation of Decision

An heuristic-based method-specialization decision has an associated cost to compute the utility function used in the heuristic decision. While making an accurate decision is ideal, accuracy usually comes at a cost of increased calculation. Thus a strategy deciding between accuracy and speed needs to be taken by the compiler.

The aspects to consider when making a method specialization decision are highlighted in Section 8.3.2.

### 8.4.3 Performing the Transformation

Performing method specialization involves copying a method, making modifications to the method, and then sending it to the optimization subsystem. Copying the method is not difficult as the unoptimized bytecode, or unoptimized machine code, just needs to be copied.

After the specialized method is created, the code needs to be linearly searched for the virtual call site instructions that target the newly specialized method (unless the virtual machine provides a constant time lookup). These call sites are then devirtualized by replacing the virtual call instruction with a direct call instruction to the specialized method.

The largest cost of performing method specialization is sending the method to the optimization subsystem. The caller must have all appropriate optimizations run on it, particularly inlining. Measuring this cost is difficult because it depends upon the method and the number of optimizations that can be done on it. Most modern compiler architectures estimate this cost by using the size of the method.

### 8.4.4 Class Loader Checks

When a new class is loaded after method specialization has been performed, the class loader needs to build the virtual method table of the new class taking into consideration that it may inherit a specialized method. Performing this check requires the compiler to keep track of the specialized target methods set for each specialized method, and then compare that set with all overriding method definitions in the new class.

This cost should not be very restrictive because method specialization is done only after the program has run long enough to determine which methods are frequently executed, which also means the majority of classes should have been loaded

already. But even if a class is loaded after performing method specialization, the check would involve going through every specialized method inherited by the newly loaded class, and checking for overriding declarations of any method in the specialized method's specialized target methods set. This lookup can be done in constant time by using efficient data structures for method lookup (most modern compilers implement this). Using this information, the class loader can select either the specialized method or the original method for the method's virtual method table.

## 8.5 Benefits of Performing Method Specialization

This framework offers the following benefits over previous implementations:

- *Dynamic class loading is explicitly handled*

  This framework adds functionality to the class loader to explicitly handle dynamic class loading. Explicit handling of dynamic class loading allows method specialization to be safely done on classes that are not known at compile time.

- *No additions to the language*

  Automatic program specialization offers method specialization for Java, but requires an extension to the Java language and requires the programmer to decide where and what to specialized. This framework requires no such extension to Java, and takes care of the specialization decisions itself.

- *More sophisticated optimization selection*

  Implementing method specialization in a dynamic compiler also gives the compiler information about the executing program. This in turn allows the compiler to select the specialization targets based on the program execution.

- *Adaptation to contextual changes*

  Long running programs can go through contextual changes. They may execute one portion of code for a period of time, and then move to a different portion of code. Performing method specialization in a dynamic compiler allows the compiler to identify these contextual changes and then adaptively react by specializing the new portions of code while deciding whether or not to discard the older specialized methods that are no longer frequently executed.

## 8.6 Summary

This new method specialization framework provides additional benefits and handles aspects that previous method specialization frameworks do not handle. The author believes that the explicit handling of dynamic class loading and the ability to implement it in a dynamic compiler making use of online profiling information, make this framework an excellent choice for the implementation of method specialization in Java.

# Chapter 9

# Empirical Study of Devirtualization Opportunities

The success of the proposed framework for method specialization depends on the existence of opportunities to specialize methods in Java programs. This chapters reports the results of a method specialization opportunity study using the industry-standard SPECjvm98 [46] and SPECjbb2000 [45] benchmark suites, hereby referred to as the "SPEC Java Benchmarks." More specifically, this study seeks to to answer the following questions:

- How many static opportunities for method specialization exist in these benchmark suites?
- What is the distribution of these opportunities throughout the program?
- How many call sites can be devirtualized in a single specialized method?
- How many devirtualizable call sites are inlinable?
- What are the sizes of the methods that can be specialized?
- Would a more liberal inlining strategy create a significant number of new method specialization opportunities?

The following sections given an overview along with a discussion of the SPEC benchmarks tested, the results collected from those tests, and a summary of the information collected.

## 9.1 Benchmark Description And Discussion

Both SPECjvm98 and SPECjbb2000 benchmark suites are written to test the execution speed of Java programs. The SPECjvm98 benchmark suite consists of 8 benchmarks that test client-side Java programs. SPECjbb2000 (Java Business Benchmark) evaluates the performance of server-side Java programs, emulating a 3-tier system (client interface, server logic, and database).

Table 9.1 gives a short description of each of the benchmarks included in the SPECjvm98 and SPECjbb2000 suites.

| Benchmark | Description |
|---|---|
| 201_compress | A Lempel-Ziv (LZW) compression program. |
| 202_jess | Java Expert Shell created by NASA. |
| 209_db | Performs multiple database functions on memory resident database written by IBM. |
| 213_javac | The Sun Microsystems Java compiler from the JDK 1.0.2. |
| 222_mpegaudio | Decompresses audio files that conform to the ISO MPEG Layer-3 audio specification. |
| 227_mtrt | A dual-threaded program that ray traces an image file. |
| 228_jack | A Java parser generator that is based on the Purdue Compiler Construction Tool Set (PCCTS). |
| SPECjbb2000 | A Java program emulating a 3-tier system with emphasis on the middle tier. |

Table 9.1: Description of the SPECjvm98 and SPECjbb2000 Benchmarks

We have chosen the SPEC benchmarks because they are widely used by industry and academic groups to perform benchmarking of Java programs. However, the SPEC Java benchmark suites are not ideal candidates for measuring method specialization opportunities. More specifically, method specialization relies on deep class hierarchies that make use of inheritance in order to find opportunities. Most of the benchmarks included in SPECjvm98 are fairly simple and do not take advantage of the object-oriented functionality provided by Java. Additionally, the simplicity of the benchmarks do not reflect the complexity of average programs. However 213_javac and SPECjbb2000 are the most complex programs of the group and therefore should be examined most closely.

An analysis done by Shuf *et al.* [113] shows that the SPEC Java benchmarks contain a small percentage of hot fields and methods. While this is characteristic of executing programs, if the programmer has not implemented the hot path of the

program in a way that will make use of method specialization, few opportunities would be present.

Dynamic class loading is also not well reflected in the SPEC Java benchmarks. All classes are loaded very early in program execution. The front-loading of most classes makes it difficult to test the performance of optimizations that support dynamic class loading because no class hierarchy changes occur later in program execution.

Regardless of these shortcomings, we have chosen to use the SPEC Java benchmarks because they are the most widely used benchmarks for performance measurement, and also because there is no other benchmark program that is both widely used and complex enough. Section 12.2 discusses the need for testing of more complex benchmarks.

## 9.2 Experimental Setup

The following sections describe our modifications and configuration to the Jikes Research Virtual Machine to perform our experiments.

### 9.2.1 Jikes Research Virtual Machine Modifications

While the Jikes Research Virtual Machine is a large and mature project, not all the required functionality for collecting information was available. The following sections describe our additions and modifications that were necessary to add the required functionality to the Jikes RVM.

#### Static Identification of Method Specialization Opportunities

In order to determine the number of method specialization opportunities that exist in the SPEC Java benchmarks, a new optimization phase needed to be written to count these opportunities. Our implementation uses the Jikes RVM optimizing compiler framework by creating a new optimization which subclasses the OPT_CompilerPhase class.

The optimization is called through a method called *perform*, which is passed the intermediate representation of the method being compiled as a parameter. The IR is walked through line by line, where it searches each instruction to determine if it is a call site where the receiver object is the *this* object.

When the optimization encounters this situation, it then obtains the target method of the call site and then checks to see if the method has been overridden. We perform this check because if the method has not been overridden, then the call site can be devirtualized without method specialization. If the method has been overridden, a potential candidate for method specialization has been identified and information is recorded about the call site and related data structures (Section 9.3).

This new optimization phase is then registered with the compiler framework so that it is aware of the new optimization. This optimization can now be configured to run at any specified optimization level and on any specified intermediate type (HIR, LIR or MIR). We propose that this optimization be performed at optimization level 2 because it currently requires SSA to determine which call sites are dispatched on the *this* object.

This optimization is also registered with VM_Callbacks, an event handling framework that calls registered methods upon specified events. This registration is required by the optimization to print out a report of the collected information upon completion of the program.

Upon completion of the program, the VM_Callbacks framework calls a method in the optimization which prints out all the collected information to a file.

### 9.2.2 Jikes Configuration

The Jikes RVM has a built in framework for performance and profiling benchmarks. In our experiments, we used the following options with the Jikes *RunSanityTest* command-line program to collect our data:

- *-test SPECjvm98*: This option tells the command to test the SPECjvm98 benchmark suite. To test the SPECjbb2000 benchmark suite, use the option *-test SPECjbb2000*.

- *-images images*: This option states that we have an existing compiler build image in the *images* subdirectory. The framework can be set up to do a fresh build of the Jikes RVM. However, this significantly increases the time it takes to complete the run.

- *-configuration FullAdaptiveSemiSpace*: This option specifies the configuration that the Jikes RVM was built with. *Full* means compile the Jikes RVM with the optimizing compiler with the highest optimization level. *Adaptive* states

that when the Jikes RVM is run, it will use the Adaptive compilation subsystem to create executable code, as opposed to the optimizing compiler. Our optimization requires the Adaptive subsystem as it contains an inlining oracle which makes the decisions as to whether to inline a call site. Finally *SemiSpace* states that the SemiSpace garbage collection system should be used.

- *-nobuild*: This option requires the *RunSanityTest* command to use a pre-build Jikes RVM image from the directory specified by the *images* parameter.

- *-results resultsDir*: This specifies the directory where the results of the run are to be written. In this case the *resultsDir* subdirectory.

- *-rc-args "-X:aos:enable_recompilation=false -X:aos:initial_compiler=opt -X:irc:O2 -X:irc:inline=false"*: These are options that are passed to the Adaptive Compilation Subsystem. Disabling recompilation ensures that methods are only compiled once. *-X:aos:initial_compiler=opt* states that the optimizing compiler is to be used for the initial compilation of a method. *-X:irc:O2* states that the initial runtime compiler (in this case the optimizing compiler) is to compile methods at optimization level 2. Finally, the *-X:irc:inline=false* option says that the initial runtime compiler is not to perform inlining.

These options allow method specialization to be run on each method when it is loaded by the virtual machine. Inlining was turned off to make sure we did not inflate the number of method specialization opportunities.

## 9.3 Static Devirtualization Opportunities

Static data was collected from benchmark suites to determine how many opportunities for devirtualization were present in benchmark programs. Section 9.3.1 describes the statistics that were collected, section 9.3.2 gives the static numbers that were collected from the SPECjvm98 [46] and the SPECjbb2000 [45] benchmark suites.

### 9.3.1 Static Specialization Empirical Information Collected

The following sections describe in detail each of the statistics that were collected from each benchmark suite, and for what reason they were collected.

### Number of Specialized Classes

The number of specialized classes (see Section 8.3.4) was collected and compared to the total number of classes loaded at run-time. Collecting only the classes loaded at run-time allows us to only measure opportunities that would be executed. Collection of these statistics were performed to determine what percentage of the classes in a program contain specializable methods, giving insight as to whether specializable methods are contained in a small subset of classes, or if they are spread out throughout the entire class hierarchy.

### Number of Specializable Methods

The number of specializable methods compared with the total number of methods loaded at run-time was collected to get a measurement of how many opportunities for devirtualization are available in each benchmark. The higher the number of methods that can be specialized, the fewer virtual calls have to be made.

However this measurement only states that the method can be specialized to at least one class. This static measurement does not indicate the number of executions (if any) the specialized method would run if the specialization was done.

### Number of Devirtualizable Call Sites

The number of devirtualizable call sites refers to the number of virtual call sites in a benchmark that can be devirtualized as a result of method specialization. This gives a good insight to just how many call sites can be unconditionally devirtualized compared with the total number of loaded call sites in the program.

### Number of Devirtualizable Call Sites That Can Be Inlined

Out of the devirtualizable call sites from method specialization, this measurement gives the number of call sites that then can be inlined. This statistic is very important because devirtualizing call sites alone does not result in a high amount of speedup. Therefore if the call sites that can be devirtualized cannot be inlined, then the speedup will likely not be high enough to justify performing method specialization.

The inlining strategy was the same strategy as used by the static compiler in the Jikes Research Virtual Machine. This strategy is described in Section 3.6.

### Number of Specializable Methods per Class

The number of specializable methods per class measures how many specializable methods are found in each class. This is another measurement to determine how contained specialized methods are; whether several of them appear in a single class or if only a few are present.

### Number of Devirtualizable Call Sites per Specializable Method

The number of devirtualizable call sites in a specializable method determines just how many virtual call sites can be devirtualized by a single specialized method. The higher the number of call sites, the better the expected execution speedup because a single specialization will allow multiple devirtualizations and multiple opportunities for additional optimizations.

### Number of Callees per Specializable Method

In order to determine if the devirtualizable call sites in a specialized method were targeting the same method, we collected the total number of callees for each specializable method. With this information, we can compare the number of devirtualizable call sites in a method (See Section 9.3.2) with the number of callees. If the number of callees for benchmark methods are low while the number of call sites are high, then we know that most call sites are to the same target method, and are therefore duplicates.

### Size of Specializable Methods

The size of specializable methods give the size in bytes of each method that potentially could be specialized. This information is important because a method specialization optimization would make a copy of the entire method. Smaller methods would be more attractive because copying them would not result in as much code growth.

### Size of Callees

For each callee of every devirtualizable call site in a specializable method, the size was examined to help determine what impact inlining the method would have on code bloat. Additionally, this information was collected to see if having a more

liberal inlining strategy that allowed bigger methods would result in a significant increase in inlinable methods.

### 9.3.2 SPECjvm98 and SPECjbb2000 Static Devirtualization Opportunities

Static data from the SPECjvm98 and SPECjbb2000 benchmark suites was collected to measure method specialization opportunities in industry benchmark suites. Data was collected from both SPECjvm98 and SPECjbb2000 benchmark suites to compare and contrast both client-side and server-side Java programs to see if there were any major differences.

The results collected are given in the following sections. The raw data for the experiments can be found in Appendix A.

#### Number of Specialization Classes

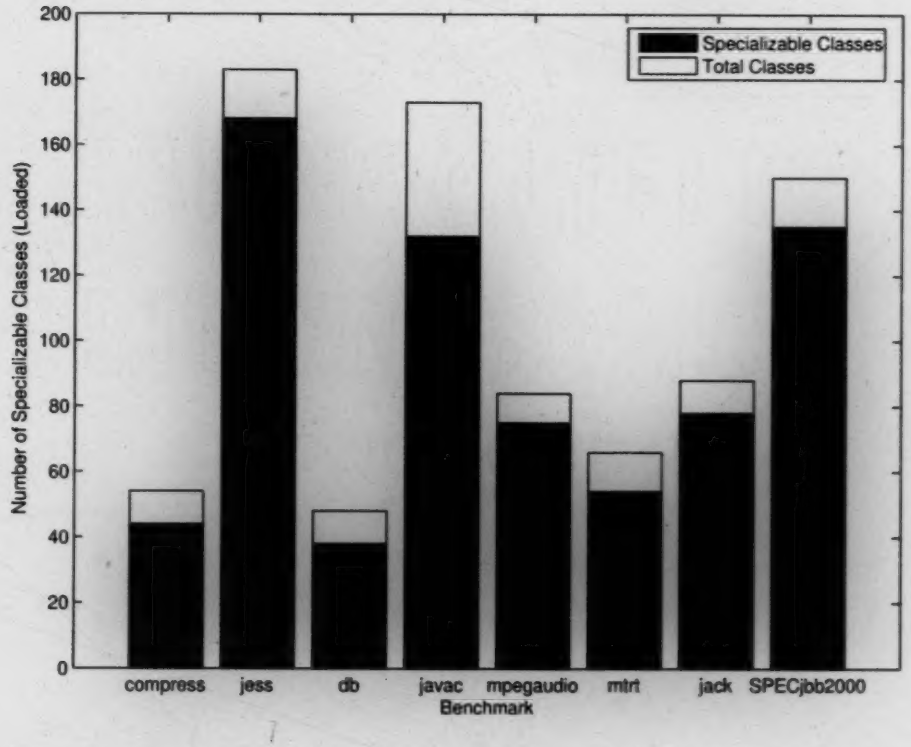


Figure 9.1: Number of Specializable Classes Compared to Total Loaded Classes in SPEC Java Benchmarks

Figure 9.1 shows the number of specialized classes compared to the total classes loaded in each benchmark (Raw data can be found in Appendix A). This data shows that most of the classes loaded contain at least one specializable method, with an average of 85% for SPECjvm98 and 90% for SPECjbb2000.

This data indicates that specialization opportunities exist throughout the whole program and are not necessarily confined to a subset of classes. On its own, this is encouraging because many classes contain specialization opportunities. However, additional information is required to find out exactly how many opportunities exist in each class.

**Number of Specializable Methods**

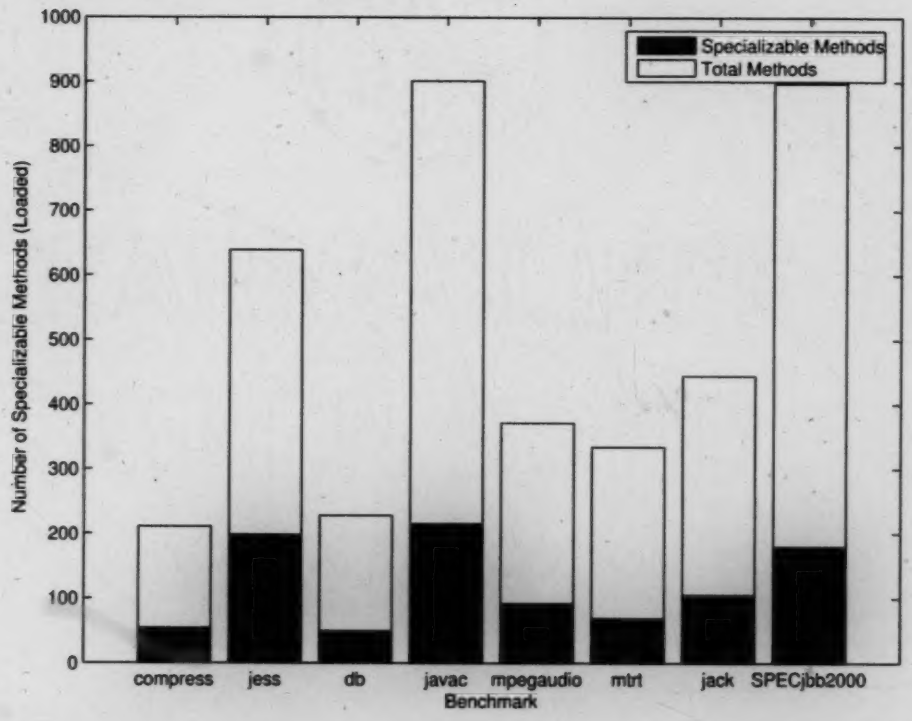


Figure 9.2: Number of Specializable Methods Compared to Total Loaded Methods in SPEC Java Benchmarks

Figure 9.2 shows the number of specializable methods compared to the total methods loaded in each benchmark (Raw data is found in Appendix A). The results show that 25% of SPECjvm98 methods and 23% of SPECjbb2000 methods are

specializable. There is not much variation in percentages, with 202_jess containing the most specializable methods at 31% and 227_mtrt on the low end containing 21%.

Finding out that 25% of all methods are specializable in these benchmark suites is encouraging given that the benchmarks have limited complexity and limited class hierarchies. However, this data must be compared with the number of call sites that can be devirtualized as a result.

### Number of Devirtualizable Call Sites

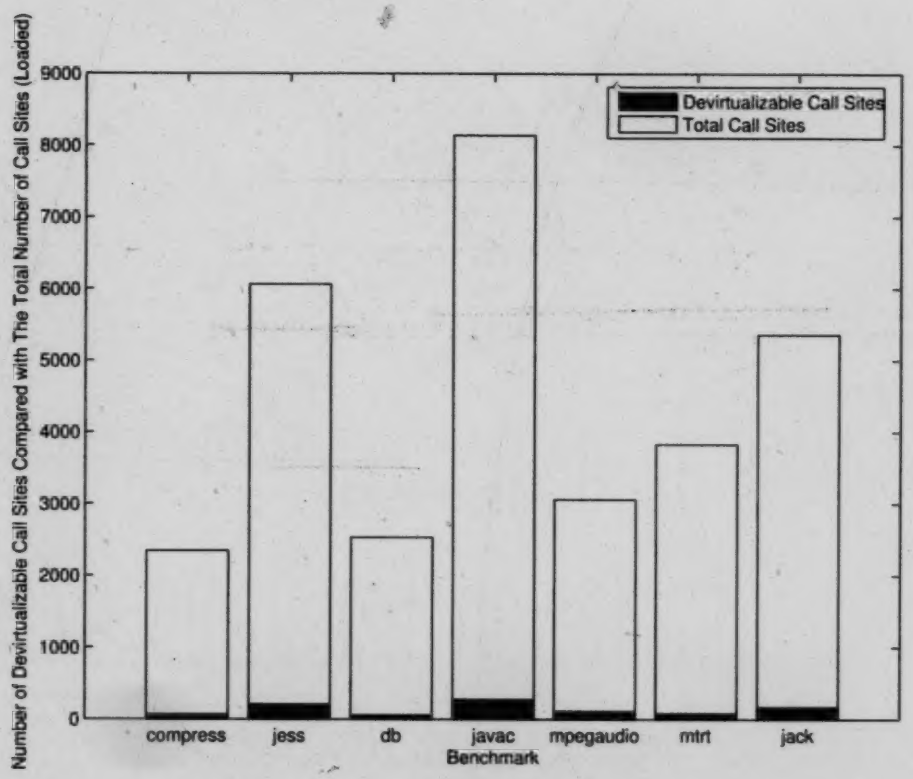


Figure 9.3: Number of Devirtualizable Call Sites Compared to Total Call Sites in SPEC Java Benchmarks

Figure 9.3 shows the number of devirtualizable call sites compared with the total number of call sites in the program (Raw data is found in Appendix A). This graph shows that the number of call sites that can be specialized is quite small compared with the total number of call sites in the program, measuring an average only 2.9%.

The percentage of call sites could be increased with better use of inheritance, as

this would increase the number of specializable methods and therefore the number of call sites available for devirtualization. However, we currently do not know the size of increase for devirtualizable call sites if inheritance were better used. Additionally, the numbers do not show the execution frequency of the call sites. Therefore a high percentage of specializable call sites is not needed for a execution speedup if those call sites are frequently executed, but a higher percentage will increase the probability of finding hot, devirtualizable call sites.

**Number of Devirtualizable Inlinable Call Sites**

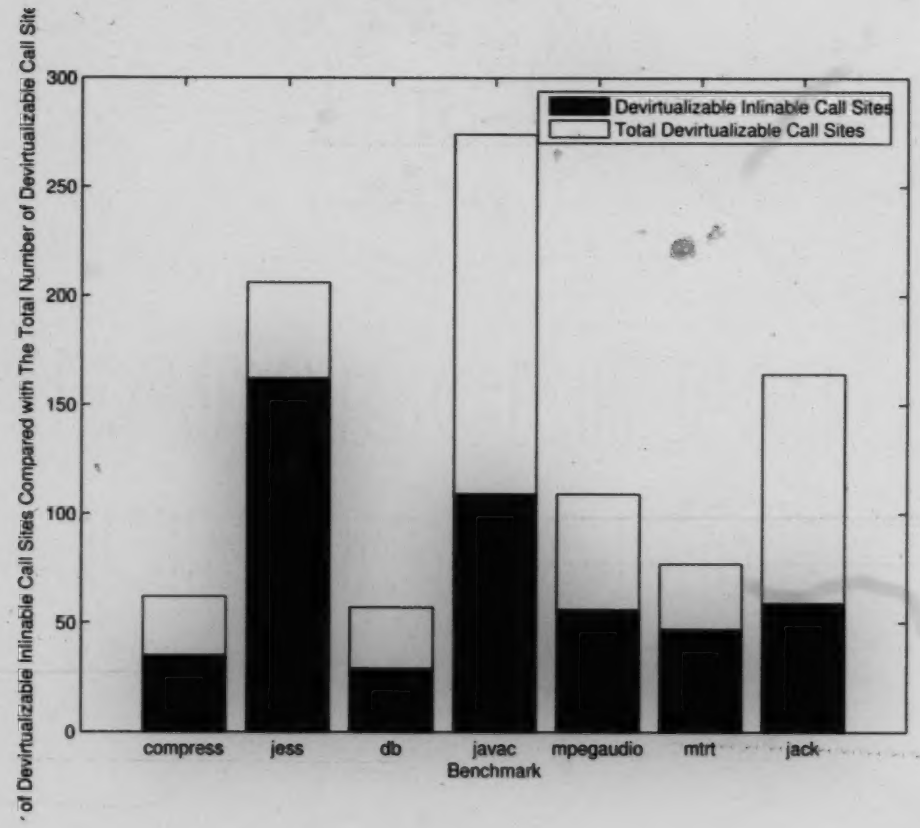


Figure 9.4: Number of Inlinable Specializable Call Sites Compared to Total Number of Call Sites in SPEC Java Benchmarks

Figure 9.4 shows a graph of the number of inlinable call sites compared with the total number of call sites that were devirtualizable from Section 9.3.2 (Raw Data can be found in Appendix A). Of the devirtualizable call sites, 70% were call sites

that could be inlined after devirtualization.

As inlining is an important optimization for improving execution speed, the strategy used by the method specialization implementation should choose from this set of call sites. While the percentage of inlinable call sites compared with devirtualizable call sites is high, the number of inlinable call sites compared with total call sites is quite low (only 2%).

### Number of Specializable Methods per Class

| Benchmark | Number of Specializable Methods per Class | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7+ | Total |
| 201_compress | 37 | 5 | 1 | 1 | 0 | 0 | 0 | 54 |
| 202_jess | 155 | 10 | 1 | 1 | 0 | 0 | 0 | 167 |
| 209_db | 30 | 6 | 1 | 1 | 0 | 0 | 0 | 38 |
| 213_javac | 96 | 18 | 6 | 4 | 2 | 5 | 1 | 132 |
| 222_mpegaudio | 63 | 8 | 3 | 1 | 0 | 0 | 0 | 75 |
| 227_mtrt | 44 | 6 | 3 | 1 | 0 | 0 | 0 | 54 |
| 228_jack | 64 | 11 | 1 | 1 | 0 | 0 | 1 | 78 |
| SPECjbb2000 | 104 | 25 | 4 | 1 | 1 | 1 | 0 | 136 |
| Total | 593 | 89 | 20 | 11 | 3 | 6 | 2 | 724 |

Table 9.2: Number of Specializable Methods per Specializable Class in SPEC Java Benchmarks

The number of specializable methods per class is shown in Table 9.2. This shows that 80% of all the classes contain only one specializable method, while 6% contain 3 or more specializable methods. This result indicates that most classes only have 1 or 2 opportunities for method specialization, while a small percentage have multiple opportunities.

As the specializable methods are spread across several classes, it would be interesting to know if there is a relationship across the 80% of classes that have only one specializable method.

### Number of Devirtualizable Call Sites per Specializable Method

Table 9.3 gives the number of call sites that are devirtualizable in each specializable method. Again while a higher number of devirtualizable call sites per specializable method would open up a higher number of optimizations opportunities, 93% of all specializable methods only have one devirtualizable call site and 98% have only 1 or 2 opportunities.

| Benchmark | Number of Devirtualizable Call Sites per Method | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11+ | Total |
| 201_compress | 49 | 4 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 54 |
| 202_jess | 193 | 4 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 198 |
| 209_db | 44 | 4 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 49 |
| 213_javac | 189 | 16 | 1 | 1 | 5 | 2 | 0 | 0 | 0 | 1 | 0 | 215 |
| 222_mpegaudio | 83 | 5 | 2 | 0 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 92 |
| 227_mtrt | 64 | 4 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 69 |
| 228_jack | 95 | 7 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 2 | 105 |
| SPECjbb2000 | 172 | 6 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 180 |
| Total | 889 | 50 | 3 | 1 | 13 | 3 | 0 | 0 | 0 | 1 | 2 | 962 |

Table 9.3: Number of Devirtualizable Call Sites per Specializable Method in SPEC Java Benchmarks

It would be ideal to be able to devirtualize multiple call sites in a single specialized method. However, it is not surprising that the majority of methods have only one or two call sites that can be devirtualized due to the constraints of a call site having a target of the *this* object and the target of the call site being overridden.

**Number of Callees per Specializable Method**

| Benchmark | Number of Callees | | | | | |
|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | Total |
| 201_compress | 50 | 4 | 0 | 0 | 0 | 54 |
| 202_jess | 194 | 4 | 0 | 0 | 0 | 198 |
| 209_db | 45 | 4 | 0 | 0 | 0 | 49 |
| 213_javac | 195 | 17 | 2 | 0 | 1 | 215 |
| 222_mpegaudio | 86 | 4 | 2 | 0 | 0 | 92 |
| 227_mtrt | 65 | 4 | 0 | 0 | 0 | 69 |
| 228_jack | 101 | 4 | 0 | 0 | 0 | 105 |
| SPECjbb2000 | 175 | 5 | 0 | 0 | 0 | 180 |
| Total | 911 | 46 | 4 | 0 | 1 | 962 |

Table 9.4: Number of Callees per Specializable Method in SPEC Java Benchmarks

Table 9.4 shows the number of callees that are target of devirtualized call sites in each specialized method. 95% of all methods have a single target, and just under 5% have 2 targets. A high percentage of specialized methods with a single callee indicates that we are likely to have duplicate call sites. However, because the majority of specializable methods have only one call site, the percentage of specializable methods that have duplicate devirtualizable call sites is small.

**Size of Specializable Methods**

| Benchmark | Size of Specializable Methods (bytes) | | | | | | |
|---|---|---|---|---|---|---|---|
| | 0-99 | 100-149 | 150-199 | 200-249 | 250-299 | 300-349 | 350+ |
| 201_compress | 0 | 31 | 7 | 9 | 1 | 1 | 5 |
| 202_jess | 0 | 156 | 13 | 14 | 3 | 3 | 9 |
| 209_db | 0 | 28 | 6 | 6 | 1 | 2 | 6 |
| 213_javac | 0 | 135 | 25 | 19 | 5 | 3 | 28 |
| 222_mpegaudio | 0 | 52 | 9 | 8 | 2 | 3 | 18 |
| 227_mtrt | 0 | 43 | 8 | 7 | 3 | 2 | 6 |
| 228_jack | 0 | 61 | 18 | 7 | 2 | 5 | 12 |
| SPECjbb2000 | 1 | 110 | 21 | 13 | 4 | 5 | 26 |
| Total | 1 | 616 | 107 | 83 | 21 | 24 | 110 |

Table 9.5: Size (bytes) of Specializable Methods in SPEC Java Benchmarks

Table 9.5 shows the sizes of all specializable methods. This data shows that the majority of all specializable methods (64%) are within the 100 to 149 byte range. Another 20% are within the 150 to 250 byte range and the largest group of size 350 or more bytes registered 11%.

This data is encouraging as the majority of specializable methods are relatively small in size. This means that several methods can be specialized without causing excessive code growth. However this data does not show which methods will cause the most amount of speedup as a result of method specialization. Therefore it is possible that methods with the highest speedup opportunities are of larger size.

**Size of Callees**

| Benchmark | Size of Callees | | | | | | |
|---|---|---|---|---|---|---|---|
| | 0-99 | 100-149 | 150-199 | 200-249 | 250-299 | 300-349 | 350+ |
| 201_compress | 31 | 15 | 6 | 0 | 1 | 2 | 2 |
| 202_jess | 154 | 21 | 8 | 1 | 1 | 2 | 4 |
| 209_db | 26 | 17 | 4 | 0 | 1 | 2 | 2 |
| 213_javac | 63 | 84 | 15 | 20 | 4 | 5 | 17 |
| 222_mpegaudio | 48 | 24 | 6 | 1 | 1 | 2 | 9 |
| 227_mtrt | 41 | 18 | 5 | 0 | 1 | 2 | 4 |
| 228_jack | 39 | 35 | 15 | 0 | 3 | 2 | 2 |
| SPECjbb2000 | 87 | 60 | 5 | 8 | 4 | 3 | 13 |
| Total | 489 | 274 | 64 | 30 | 16 | 20 | 53 |

Table 9.6: Size (bytes) of Devirtalizable Call Site Callees in SPEC Java Benchmarks

Table 9.6 shows the sizes of the target methods for devirtualizable call sites. 80% of all callees were within the 0-150 byte range with only 6% in the 350 bytes and larger category. This is also encouraging as the majority of call sites that would be inlined are small in size.

This information was collected to determine if a more liberal inlining strategy could significantly increase the number of call sites that could be inlined. The current Jikes RVM inlining strategy uses a constant for the maximum size for an inlined method of 135 bytes. The data does not show how many methods are between 135 bytes and 150 bytes, but increasing the constant to 150 bytes would see that the full 80% of devirtualizable call sites could be inlined. But this does not guarantee that they would actually in fact be inlined.

### 9.3.3 Discussion of Static Opportunity Numbers

Firstly, when comparing the number of specialized methods with the number of devirtualizable call sites, the data shows that a large percentage of methods are specializable (at 25%) while the number of call sites that can be devirtualized is very low (at about 3%). These results indicates that the small amount of call sites that are devirtualizable as a result of method specialization, are spread out across several methods. As a consequence, method specialization is less attractive because it would need to be performed on several methods to devirtualize these call sites. This is costly in terms of compilation time as well as code growth.

Comparing the number of devirtualizable call sites with the number of devirtualizable call sites, the percentage of good candidates drops to 2%. Again, this number may be raised with better use of inheritance. However it is not known how large the effect would be. However given that the most complex benchmarks, 213_javac and SPECjbb2000, have only 3.6% and 2.6% devirtualization opportunities respectively, it appears that the gain in the number of opportunities may not be large.

Comparing the number of devirtualizable call sites with the number of callees per specializable method shows that a small number of the call sites are duplicates. As 93% of all specializable methods have one devirtualizable call site but 95% of specializable methods have only a single callee for devirtualizable call sites, 2% of specialized methods have at least one duplicate call site.

## 9.4 Summary

The static numbers of the SPECjvm98 and SPECjbb2000 benchmarks are not very promising for method specialization. The total number of call sites that can be devirtualized from method specialization is below 3% and the number of those call sites that can be inlined drops the number to 2%.

These numbers are significantly lower than selective specialization which was able to devirtualize up to 66% of all virtual call sites, and customization which was able to devirtualize up to 61%. The exact reason for this large discrepancy is not known at this time. We believe that testing other larger, more complex, object-oriented benchmarks will close this gap slightly. However, the gap is still quite large and should be examined further.

These numbers also must be taken in context with the dynamic numbers to determine if the possible devirtualizable call sites are frequently executed. With this information, the potential speedup from method specialization can be more accurately measured.

# Chapter 10

# Related Work

In this chapter we talk about related research in the areas of call site devirtualization and method specialization. Because call site devirtualization is important for creating opportunities for other optimizations such as inlining, several techniques have been introduced to eliminate dynamic dispatches from pure object-oriented programming languages. Much research has been done in the area of statically-compiled systems with offline profile feedback information, and a significant amount of research has been performed for dynamically-compiled systems. While work has also been done in the area of devirtualization in dynamic class loading language, online method specialization in dynamic class loading languages does not appear to have been studied.

## 10.1 Extant Analysis

Sreedhar introduced the concept of *Extant Analysis* [115] which allows interprocedural optimization in the presence of dynamic class loading. Given a set of classes called the closed world set, offline static analysis is performed to partition object references into 2 sets: *unconditionally extant* or object references that are guaranteed to be in the set of closed world classes, and *conditionally extant* or object references that may point to a class that is not in the closed set of classes. Based on this information, static optimizations that can be applied to the first unconditionally extant set are guaranteed to remain correct, even after dynamic class loading.

Consider the example in Figures 10.1 and 10.2, which are based on the example given in [115]. The classes shown created the *closed world* set of classes which is a set of classes. By using extant analysis, we determine if object references refer to only this closed set of classes, or if they could refer to classes that are outside of

this set. This is done by performing data-flow analysis on the closed world classes. Using this data-flow analysis, we can determine that method call site *b.bar()* in class A is extant because the runtime type of the receiver object *b* is guaranteed to be a type in the closed world of classes. However method call site *c.bar()* in class A cannot be assumed to be in this set, as variable *c* is a parameter to the public method *func(C)* which can be called from outside of the closed world. Thus we must assume that there exists the possibility of a class *D* extending class *C* that contains an overriding method *bar()*, that is passed as a parameter to method *func(C)*. Thus any optimization done on this object reference must be guarded with a runtime test to check the dynamic type.

```
public class A {
    public static void func(C c) {
        ...
        B b = new B();
        b.bar();
        c.bar();
    }
}

public class B {
    public void bar() { ... }
}

public class C extends B { ... }
```

Figure 10.1: A Sample Pseudo-Java Class Hierarchy Illustrating Extant Analysis, After [115]

Extant Analysis also works to identify methods that are candidates for method specialization. By finding object references that are conditionally extant, it is possible to produce a method that is specialized towards a particular class. For example in Figure 10.1, method *A.func(C)* could be specialized for the closed world of classes. A new method *A.func'(C)* is specialized for the receiver type of variable *c* being in the closed world, while the original method is used for all other cases. Method selection is then done by inserting an Extant Safety Test (EST) which checks the runtime type of the object reference and calls the appropriate method.

Extant Analysis performs well as it is a static optimization that incurs no runtime cost. Additionally, testing showed that for the SPECjvm98 benchmarks, 97.9% of method calls are within the closed world allowing a tremendous amount of optimization [115]. However specialization, as a static optimization, is hindered because

```
public class A {
    public static void func(C c) {
        ...
        B b = new B();
        b.bar();
        c.bar();
    }
    public static void func'(C c) {
        ...
        B b = new B();
        b.bar();
        B.bar(); // Devirtualized method call site
    }
}

public class B {
    public void bar() { ... }
}

public class C extend B { ... }

public class D {
    public static void something() {
        ...
        if(EST(c1)) // Returns true if c1 is an extant object
            A.func'(c1); // Specialized method
        else
            A.func(c1); // Original method
    }
}
```

Figure 10.2: A Sample Pseudo-Java Class Hierarchy Illustrating Extant Analysis After Specialization, After [115]

it suffers from code bloat. Method specialization tends to work best as a dynamic optimization where the compiler is able to identify hot portions of code and create specialized methods based on this information to restrict code bloat.

## 10.2 Class Hierarchy Analysis

*Class Hierarchy Analysis* (CHA) [50] is a technique that uses knowledge of a program's inheritance graph to perform optimizations such as statically-binding dynamic method calls, and determining the set of classes that are the target of a particular method.

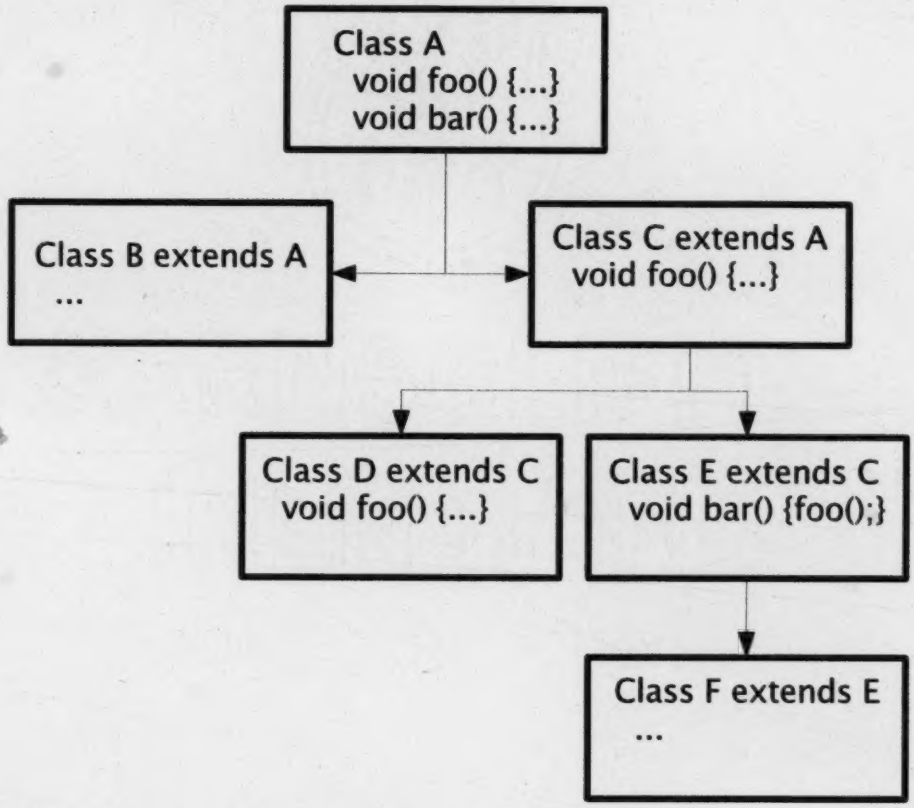


Figure 10.3: A Sample Class Hierarchy

Figure 10.3 shows a sample class hierarchy. Class *E* defines a method *bar()* that includes a method call site to method *foo()*. No subclass of class *E* contains an overriding declaration of method *foo()*. Thus there is only one target class for the

call site (*C.foo()*), and therefore a direct call can be used instead of a virtual call.

CHA is quite effective and can speedup execution between 23% to 89% and reduce code size by 12% to 21% for dynamically-typed programs [50]. CHA is also powerful when used in combination with profile-guided receiver class prediction, producing speedups between 45% to 410% [75]. This speedup was attributed to the chemistry between the optimizations as the compiler can use CHA to determine monomorphic call sites and then fall back on profile-guided receiver class prediction for polymorphic call sites.

CHA is generally used on non-dynamic class loading languages because a complete knowledge of all classes is needed to create the class hierarchy. Because Java provides dynamic class loading, the entire class hierarchy can never be known as a new class can be loaded at any point during the program execution. Thus Java compilers that use CHA utilize a class hierarchy that is current for a given program point, but all optimizations must allow for additional classes to be loaded at a later point.

## 10.3 Profile-Guided Receiver Class Prediction

Profile-guided receiver class prediction [65, 73, 75] utilizes run-time type feedback to determine what is the likely type of the receiver for a call site. In the system implemented by Holzle and Ungar [75], the compiler creates an instrumented version of the program that collects profiling information about the types of receivers at all call sites. This profiling is then fed back to the compiler to generate optimized code. The optimized code attempts to predict the receiver of dynamically-dispatched call sites based on the profile information given.

Figure 10.4 gives a sample class hierarchy and Figure 10.5 gives a call site and what the corresponding optimized code looks like. This example was first presented by Hölzle *et al.* in [75].

The optimized code optimizes for receiver objects of type *CartesianPoint*. This in turn allows the virtual call to be statically bound and inlined, while protecting the program in the instance that the *Point* object was in fact not a *CartesianPoint*.

While this in turn speeds up the execution by a factor of 1.5 [75], Profile-guided receiver class prediction can slow down execution if the profile taken doesn't yield good results. As a result, Grove *et al.* [65] examined 4 benchmarks to determine if the profiles gathered from the benchmarks were:

```
public abstract class Point {
    float abstract getX();
    float abstract getY();

    public abstract float getDistance(Point paramPoint);
}

public class CartesianPoint extends Point {
    float x, y;
    public float getX {
        return x;
    }

    // Other methods omitted
}

public class PolarPoint extend Point {
    float rho, theta;
    public float getX() {
        return rho * cos(theta);
    }

    // Other methods omitted
}
```

Figure 10.4: A Sample Pseudo Java Class Hierarchy from [75]

```
Point p = getSomePoint();
float x = p.getX();

if(p.instanceOf(CartesianPoint)) {
    // Inline CartesianPoint method
    x = p.x;
} else {
    // Original virtual call
    x = p.getX();
}
```

Figure 10.5: A Call Site and the Corresponding Optimized Code from [75]

- Strongly Peaked: Do the call sites have a small number of dominant receiver types?
- Stable Across Input: Do the class distributions change a significant amount across different input for the same program?
- Stable Across Version: Does the profile remain valid as the program goes through its development?
- Stable Across Programs: Is there a significant difference in receiver types in a library that is shared across multiple programs?

Grove et al. recorded that the profile information collected indeed was strongly peaked and stable across input and version for their particular benchmarks. However they were unable to measure the stability across programs, leaving a significant question unanswered. While profile-guided receiver class prediction does indeed provide excellent speedup for statically-compiled languages, it's use in programming languages that allow dynamic class loading is limited. When a subclass, that would benefit from the optimized code, is dynamically loaded at run-time it will not make use of the optimized code because the class test provided needs to change and include the newly loaded class. This is quite tedious and requires recompilation of each specialized method each time a subclass is dynamically loaded.

## 10.4 Code Patching

Code patching [39, 76, 117], is a devirtualization technique that attempts to avoid recompilation of methods when inlining decisions are invalidated. If a call site only has a single target, the compiler creates two separate portions of code. The first portion of code contains devirtualized and inlined call sites without any guarding while the second portion contains the original virtual call sites. The code is then set up such that the inlined code is always executed and the original code is never executed. In the case of a dynamic class load that invalidates the devirtualization, a branch instruction to the original code is written over the first instruction of the devirtualized code. Thus execution goes to the slower, but correct, original portion of code.

Figures 10.6 and 10.7, which are copied from [76] for illistrative purposes, show the corresponding PowerPC instructions of the before and after situations for code

```
inlined_code:
    add 5, D0
    add 4, D1
    ...
code_after_inlined_code:
    ...
original_call:
    lwz r1, (obj) // load class pointer
    lwz r2, offset(r1) // load method pointer
    lwz r3, offset(r2) // load code address
    mtctr r3 // move address into condition register
    blr ctr // dynamic method call
    b code_after_inlined_code // Branch to the next instruction
```

Figure 10.6: Assembly Code Before Dynamic Class Loading From [76]

```
inlined_code:
    b original_call // Jump to the virtual call code
    add 4, D1 // No longer executed
    ...
code_after_inlined_code:
    ...
original_call:
    lwz r1, (obj) // load class pointer
    lwz r2, offset(r1) // load method pointer
    lwz r3, offset(r2) // load code address
    mtctr r3 // move address into condition register
    blr ctr // dynamic method call
    b code_after_inlined_code // Branch to the next instruction
```

Figure 10.7: Assembly Code after Dynamic Class Loading that Invalidates Inlining Decision From [76]

patching. Figure 10.6 shows how virtual call sites are devirtualized and the target method inlined (code is indicated by the label "inlined_code"). Figure 10.7 shows the changes made after dynamic class loading that invalidates the devirtualization. As shown in the example, the first line of the inlined method is replaced by a branch instruction to the original virtual call instructions.

Code patching gives a speedup of approximately 13% with SPECjvm98 and SPECjbb2000. When code patching is combined with other devirtualization techniques such as preexistence, speedups reach approximately 16%.

## 10.5 Pre-existence Analysis

Pre-existence analysis [51] is another technique used to devirtualize method call sites in dynamic programming languages (such as Java) and was designed to avoid costly On-Stack Replacement [74] when dynamic class loading invalidates previous inlining decisions. Pre-existence identifies if a object reference pre-exists, which is defined as being allocated before the execution of a particular method. If an object is allocated prior to the execution of the method, the object must be in the set of classes extant in the program at the start of the method.

```
void foo(O o) {
    O o2;
    ...
    o2 = o;
    o2.bar(); // Object o2 is pre-existing
    O o3 = new O();
    o3.bar(); // Object o3 is not pre-existing
}
```

Figure 10.8: An Example Illustrating Pre-existence After [51]

Take the example found in Figure 10.8 that is based on the example given in [51]. Looking at the call site *o2.bar()*, the receiver object *o2* is assigned the value of the parameter *o*. As variable *o* is a parameter, the object must have been allocated before the execution of method *foo(O)*. Thus we can inline the call site *o2.bar()* without a guard and if a new class that invalidates the inlining decision is loaded during method *foo(O)*, the method can safely execute until the end of execution. Once the method has finished executing, the method must be recompiled to correct the inlining optimization.

The call site *o3.bar()* is a different situation. Variable *o3* is allocated during the execution of method *foo(O)* and therefore does not pre-exist. Therefore the call site could be devirtualized and inlined but if a class were loaded during the execution of method *foo(O)*, then on-stack replacement must be used to correct the now incorrect inlining decision.

Results showed that for the benchmarks tested, approximately half of the virtual call sites had receiver objects that were proven to be pre-existing. While pre-existence eliminates the need for costly on-stack replacement, recompilation of methods is still required when dynamic class loading invalidates inlining decisions which can be costly.

## 10.6 Thin Guards

The concept of Thin Guards [13] was developed as an alternative to class tests [32, 65] and method tests [51]. Generally speaking, Thin Guards provide 2 benefits over traditional class and method guard tests:

1. Thin Guards are generally more efficient than class or method tests.
2. Thin Guards test a more general condition, allowing more optimization for a single guard.

Thin Guards provide an efficient runtime test to allow optimistic assumptions to be performed in the presence of dynamic class loading. Optimistic assumptions are facts about the executing program that:

1. Are currently true.
2. Are unlikely to change in the near future.
3. Enable additional optimization to be performed.

Thin Guards take these optimistic assumptions and map them to a condition bit. An example is testing if dynamic class loading has happened after a specific point in program execution. If no dynamic class loading has been performed, the condition bit is set to false. If dynamic class loading has happened, regardless of the class loaded the condition bit is set to true. Figure 10.9 shows an example of an opportunity for inlining while Figure 10.10 shows how Thin Guards can be used

to perform an optimization and protect against dynamic class loading that may invalidate the optimization. Both Figures 10.9 and 10.10 are based on the example found in [13].

```
class A {
    public int func() {
        A a = getSomeNewAorB();
        B b = getSomeB();
        return a.getX() + b.getY();
    }
    public int getX() {
        return 1;
    }
    public int getY() {
        return 2;
    }
}

class B extends A { ... }
```

Figure 10.9: Sample Java Pseudo Code Demonstrating an Opportunity for Inlining, After [13]

```
class A {
    public int func() {
        A a = getSomeNewAorB();
        B b = getSomeB();
        if(noClassloadingHasOccured) {
            return 3;
        } else {
            return a.getX() + b.getY();
        }
    }
    public int getX() {
        return 1;
    }
    public int getY() {
        return 2;
    }
}

class B extends A { ... }
```

Figure 10.10: Sample Java Pseudo Code Demonstrating Using Thin Guards For Inlining, After [13]

As Figure 10.9 shows, the call sites *a.getX()* and *b.getY()* are currently monomorphic, meaning that they currently only have one target class, but dynamic class loading could invalidate this at a later point. Thus we perform speculative inlin-

ing and protect the execution by placing a Thin Guard. Figure 10.10 shows what method *func()* looks like after the guard was inserted. We now have a test that checks a condition to see if any classes were loaded after performing the optimization. If not, then the devirtualized, inlined, and constant propagated folded code is executed while the original virtual calls are executed if a new class has been loaded.

Thin Guards showed impressive performance when comparing it with an ideal performance level where unsafe class hierarchy analysis was used (inlining with no guard). When no guard was used to protect optimizations from dynamic class loading, Thin Guards achieved between 70% and 92% of this ideal performance on the SPECjvm98 benchmark suite. This showed that the overhead of eliminating penalties of dynamic class loaded with Thin Guards was reasonably small, while providing benefits from speculative optimization.

Unlike class and method tests, Thin Guards are only effective for currently monomorphic call sites, as they cannot distinguish between receiver types at call sites. Additionally if the condition bit were the same as the one used in the previous example, then all optimized code is not executed in the event of a single dynamic class loading, regardless of the validity of the optimized code. However this can be avoided by providing several condition bits for particular classes, but this increases the overhead of performing the test. The authors acknowledge that understanding the ideal number of conditions for an executing program requires more research.

# Chapter 11

# Conclusions

Method specialization is an important optimization for eliminating virtual call sites and opening up opportunities for other compiler optimizations. This thesis examined previous method specialization research, and its applicability to dynamic compilation environments that handle dynamic class-loading.

Customization is a brute force method specialization technique that eliminates all virtual call sites to the "this" receiver object at the expense of enormous code growth and compilation time. Customization also does not handle the problem of creating multiple copies of a method when only one method would suffice, and only specializes on the receiver object for call sites.

Selective specialization handles the code growth problem of method specialization much better than customization does. By selectively choosing only frequently executed methods to specialize, unnecessary specialization is avoided. Selective specialization also adds the functionality of specializing not only on the receiver object at a call site, but also on parameter object-types. While this technique certainly improves on the customization technique, it does not handle dynamic class-loading and the heuristic is not ideal for a dynamic compilation environment.

Automatic program specialization was designed for the Java programming language and allows the programmer to define where and what to specialize. While this technique gives complete control to the programmer, it requires an extension to the language; it sacrifices the ability to adapt to different run-time values and context changes; and it does not explicitly handle dynamic class-loading.

Given that the previous method specialization techniques do not satisfy the requirements of being suitable for a dynamic compilation environment and explicitly handling dynamic class-loading, we presented a new method specialization compiler

optimization framework that handles both of these requirements. Our framework uses on-line profiling information to first identify which call sites will benefit from devirtualization. Then it uses receiver-type profiling information to identify which object-type the method should be specialized to. Once identified, the framework shows how to perform method selection using virtual method tables, and gives the additional checks that need to be inserted into the class loader to ensure correct execution upon dynamic class-loading.

We also identified the aspects of the program that should be examined when making method specialization decisions. These aspects can be used to create a heuristic for method specialization decisions.

Finally, we collected static information regarding method specialization opportunities in the widely-used SPECjvm98 and SPECjbb2000 benchmarks. The numbers collected show that only a small percentage of all call sites in these benchmarks can be devirtualized as a result of method specialization. More research needs to be performed to identify if this is related to implementation of the benchmarks, or some other problem.

# Chapter 12

# Future Work

## 12.1 Collection of Dynamic Execution Information for Method Specialization Opportunities

As identified in Chapter 9, it is important to know if the sites where the is an opportunity for method specialization are frequently executed. As programs usually spend the majority of execution time in a small segment of the program, having multiple method specialization opportunities will not result in significant speedup if these opportunities are not in a hot path.

Therefore, tests need to be performed to measure the dynamic execution count of call sites that can be devirtualized due to method specialization. Additionally, information regarding dynamic receiver-object types of devirtualizable call sites would also be informative as it would indicate which object-types are receiving the majority of calls.

## 12.2 Test Additional Complex Benchmarks

Chapter 9 presented the results for our investigation of method specialization opportunities for the SPECjvm98 and SPECjbb2000 benchmarks. We discussed how these benchmarks were not ideal candidates for measuring method specialization opportunities due to their small size and lack of inheritance and abstraction utilization.

It would be beneficial to collect method specialization opportunity data on larger, more complex, benchmarks to determine if more opportunities exist in programs that make use of inheritance and abstraction.

## 12.3 Removal of Specialized Methods

As mentioned before, long running programs can go through context changes in execution. These context changes can render specialized methods useless as optimizations are only effective if the method is executed. Also as method specialization can add to code bloat, a program may not be able to perform additional method specializations after a context change due to space constraints.

The ability to chose previous method specializations and eliminate them would allow the compiler to free up space and therefore create new specialized methods targeted at the new execution behavior. It would be beneficial for a compiler to be able to identify candidates for deletion based on context-sensitive profiling information. This in turn would allow a method specialization framework to adapt to program changes without having to worry about space constraints.

## 12.4 A Heuristic for Making Method Specialization Decisions

The ability for a dynamic compiler to make smart optimization decisions is very important as the cost for optimization is paid at run-time. Optimization decisions must provide speedup that is greater than the cost it takes to perform the optimization. The aspects that a compiler needs to take into consideration when making such a decision is highlighted in Section 8.3.2.

With this information the compiler must made method specialization decisions. As the compiler wants to make the best optimization decisions based on this information that balances between speedup, cost, and code growth, research into what heuristic provides the best results is needed.

This research requires a working method specialization implementation along with several test cases with which to test the heuristic on.

# Bibliography


[1] Gnu compiler collection. http://gcc.gnu.org, 2005.

[2] The multijava project. http://multijava.sourceforge.net, 2005.

[3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[4] Gerald Aigner and Urs Hölzle. Eliminating virtual function calls in C++ programs. *Lecture Notes in Computer Science*, 1098:142–??, 1996.

[5] F. E. Allen and J. Cocke. A catalogue of optimizing transformations. *Design and Optimization of Compilers*, pages 1–30, 1972.

[6] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The jalapeno virtual machine. In *IBM Systems Journal*, number 1, pages 211–238, 2000.

[7] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–11. ACM Press, 1988.

[8] Bowen Alpern, C. Richard Attanasio, John J. Barton, Anthony Cocchi, Susan Flynn Hummel, Derek Lieber, Ton Ngo, Mark F. Mergen, Janice C. Shepherd, and Stephen Smith. Implementing jalapeno in java. In *Conference on Object-Oriented*, pages 314–324, 1999.

[9] Bowen Alpern, Maria Butrico, Anthony Cocchi, Julian Dolby, Stephen Fink, David Grove, and Ton Ngo. Experiences porting the jikes rvm to linux/ia32.

[10] Bowen Alpern, Anthony Cocchi, Derek Lieber, Mark Mergen, and Vivek Sarkar. Jalapeo - a compiler-supported java virtual machine for servers.

[11] Helle Markmann Andersen and Ulrik Pagh Schultz. Declarative specialization for object-oriented-program specialization. In *PEPM '04: Proceedings of the 2004 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 27–38, New York, NY, USA, 2004. ACM Press.

[12] M. Arnold. Online instrumentation and feedback-directed optimization of java, 2002.

[13] M. Arnold and B. Ryder. Thin guards: A simple and effective technique for reducing the penalty of dynamic class loading. In *Proceedings on the European Conference on Object-Oriented Programming*, 2002.

[14] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive optimization in the Jalapeño JVM: The controller's analytical model. *3rd ACM Workship on Feedback-Directed and Dynamic Optimization (FDDO-3)*, 2000.

[15] Matthew Arnold, Stephen J. Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive optimization in the jalapeno JVM. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'00)*, pages 47–65, 2000.

[16] Matthew Arnold, Stephen J. Fink, Vivek Sarkar, and Peter F. Sweeney. A comparative study of static and profile-based heuristics for inlining. In *Dynamo*, pages 52–64, 2000.

[17] Matthew Arnold, Michael Hind, and Barbara G. Ryder. An empirical study of selective optimization. *Lecture Notes in Computer Science*, 2017:49–??, 2001.

[18] Matthew Arnold, Michael Hind, and Barbara G. Ryder. Online feedback-directed optimization of java. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 111–129. ACM Press, 2002.

[19] Matthew Arnold and Barbara G. Ryder. A framework for reducing the cost of instrumented code. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 168–179, 2001.

[20] John Aycock. A brief history of just-in-time. Technical report, 2001.

[21] Marc Berndl, Ondrej Lhotk, Feng Qian, Laurie Hendren, and Navindra Umanee. Points-to analysis using bdds. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 103–114. ACM Press, 2003.

[22] Joshua Bloch. *Effective Java*. The Java Series. Addison Wesley, 2001.

[23] K. Bowers and D. Kaeli. Characterizing the spec jvm98 benchmarks on the java virtual machine, 1998.

[24] M. Burke, J. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. Serrano, V. Sreedhar, H. Srinivasan, and J. Whaley. The jalapeno dynamic optimizing compiler for java. In *Proceedings ACM 1999 Java Grande Conference*, pages 129–141, San Francisco, CA, United States, June 1999. ACM.

[25] Brad Calder and Dirk Grunwald. Reducing indirect function call overhead in C++ programs. In *Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 397–408, Portland, Oregon, 1994.

[26] C. Chambers and D. Ungar. Customization: optimizing compiler technology for self, a dynamically- typed object-oriented programming language. In *Proceedings of the SIGPLAN '89 Conference on Programming language design and implementation*, pages 146–160. ACM Press, 1989.

[27] C. Chambers, D. Ungar, and E. Lee. An efficient implementation of SELF a dynamically-typed object- oriented language based on prototypes. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 24, pages 49–70, New York, NY, 1989. ACM Press.

[28] Craig Chambers. *The Design and Implementation of the SELF Compiler, an Optimizing Compiler for Object-Oriented Programming Languages.* PhD thesis, 1992.

[29] Craig Chambers, Jeffrey Dean, and David Grove. Whole-program optimization of object-oriented languages. Technical Report TR-96-06-02, 28, 1996.

[30] Craig Chambers, Igor Pechtchanski, Vivek Sarkar, Mauricio J. Serrano, and Harini Srinivasan. Dependence analysis for java. In *Workshop on Languages and Compilers for Parallel Computing*, August 1999.

[31] Craig Chambers and David Ungar. Iterative type analysis and extended message splitting: Optimizing dynamically-typed object-oriented programs. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 150–164, 1990.

[32] Craig Chambers and David Ungar. Making pure object-oriented languages practical. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 26, pages 1–15, New York, NY, 1991. ACM Press.

[33] Craig Chambers and David Ungar. A retrospective on "customization: Optimizing compiler technology for self, a dynamically-typed object-oriented programming language". *SIGPLAN Not.*, 39(4):295–312, 2004.

[34] Bay-Wei Chang and David Ungar. Animation: From cartoons to the user interface. In *ACM Symposium on User Interface Software and Technology*, pages 45–55, 1993.

[35] Pohua P. Chang, Scott A. Mahlke, and Wen mei W. Hwu. Using profile information to assist classic code optimizations. *Software - Practice and Experience*, 21(12):1301–1321, 1991.

[36] Trishul M. Chilimbi and Martin Hirzel. Dynamic hot data stream prefetching for general-purpose programs. In *Proceeding of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 199–209. ACM Press, 2002.

[37] Jong-Deok Choi, David Grove, Michael Hind, and Vivek Sarkar. Efficient and precise modeling of exceptions for the analysis of java programs. In *Proceedings of the 1999 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 21–31. ACM Press, 1999.

[38] Jong-Deok Choi, Manish Gupta, Mauricio J. Serrano, Vugranam C. Sreedhar, and Samuel P. Midkiff. Escape analysis for java. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 1–19, 1999.

[39] Michał Cierniak, Guei-Yuan Lueh, and James M. Stichnoth. Practicing judo: Java under dynamic optimizations. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 13–26. ACM Press, 2000.

[40] C. Clifton, G. Leavens, C. Chambers, and T. Millstein. Multijava: Modular open classes and symmetric multiple dispatch for java. In *Proc. ACM'2000*, 2000.

[41] Paul Coddington, Ken Hawick, and Jesudas Mathew. Java grande benchmarking. http://www.dhpc.adelaide.edu.au/projects/javagrande/benchmarks, 2003.

[42] Charles Consel, Luke Hornof, François Noël, Jacques Noyé, and Nicolae Volanschi. A uniform approach for compile-time and run-time specialization. In *Partial Evaluation. International Seminar.*, pages 54–72, Dagstuhl Castle, Germany, 12-16 1996. Springer-Verlag, Berlin, Germany.

[43] International Business Machines (IBM) Corporation. The jikes™ research virtual machine user's guide post 2.2.2 (cvs head). http://www-124.ibm.com/developerworks/oss/jikesrvm/userguide/HTML/userguide.html, 2003.

[44] International Business Machines (IBM) Corporation. The jikes™ reserch virtual machine (rvm) website. http://jikesrvm.sourceforge.net/, 2003.

[45] The Standard Performance Evaluation Corporation. Spec java business benchmark (jbb) 2000. http://www.spec.org/osg/jbb2000, 1996.

[46] The Standard Performance Evaluation Corporation. Spec java virtual machine (jvm) 98 benchmarks. http://www.spec.org/osg/jvm98, 1996.

[47] B. de Alwis, S. Gudmundson, G. Smolyn, and G. Kiczales. Coding issues in aspectj, 2000.

[48] Jeffrey Dean, Craig Chambers, and David Grove. Selective specialization for object-oriented languages. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 93–102, 1995.

[49] Jeffrey Dean, Greg DeFouw, David Grove, Vassily Litvinov, and Craig Chambers. VORTEX: An optimizing compiler for object-oriented languages. In *Proceedings OOPSLA '96 Conference on Object-Oriented Programming Systems, Languages, and Applications*, volume 31, pages 83–100, 1996.

[50] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. *Lecture Notes in Computer Science*, 952:77–??, 1995.

[51] D. Detlefs and O. Agesen. Inlining of virtual methods. In *Proceeding of the Conference of Object-Oriented Programming System, Languages, and Applications*, pages 258–279, 1999.

[52] L. Peter Deutsch. Richards benchmark.

[53] Sylvia Dieckmann and Urs Hölzle. The space overhead of customization. Technical Report TRCS97-21, 10, 1997.

[54] Julian Dolby and Andrew Chien. An automatic object inlining optimization and its evaluation. *ACM SIGPLAN Notices*, 35(5):345–357, 2000.

[55] Karel Driesen and Urs Hölzle. The direct cost of virtual function calls in c++. In *Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 306–323. ACM Press, 1996.

[56] Evelyn Duesterwald, Calin Cascaval, and Sandhya Dwarkadas. Characterizing and predicting program behavior and its variability. *Parallel Architectures and Compilation Techniques (PACT) 2003*, 2003.

[57] Carole Dulong, Rakesh Krishnaiyer, Dattatraya Kulkarni, Daniel Lavery, Wei Li, John Ng, and David Sehr. An overview of the Intel IA-64 compiler. *Intel Technology Journal*, (Q4):15, 1999.

[58] Tzilla Elrad, Robert E. Filman, and Atef Bader. Aspect-oriented programming. *Communications of the ACM*, 44(10):29–32, October 2001.

[59] Michael Morrison et al. *Java 1.1 Unleashed*. Sams.net, third edition, 1997.

[60] P. Fradet and M. Sudholt. Aop: towards a generic framework using program transformation and analysis.

[61] Robert J. Walker Gail C. Murphy, Elisa L. A. Baniassad, Martin P. Robillard, Albert Lai, and Mik A. Kersten. Does aspect-oriented programming work? *Communications of the ACM*, 44(10):75–77, October 2001.

[62] Karthik Gargi. A sparse algorithm for predicated global value numbering. In *Proceeding of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 45–56. ACM Press, 2002.

[63] M. Golm, M. Felser, C. Wawersich, and J. Kleinoder. The jx operating system, 2002.

[64] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. The Java Series. Addison Wesley, first edition, 1997.

[65] David Grove, Jeffrey Dean, Charles Garrett, and Craig Chambers. Profile-guided receiver class prediction. In *Conference on Object-Oriented*, pages 108–123, 1995.

[66] Dick Grune and Ceriel J. Jacobs. *Modern Compiler Design*. John Wiley And Sons Canada, Ltd, 2000.

[67] Nathan M. Hanish and William Cohen. Hardware support for profiling java programs.

[68] Brinch Hansen. *Brinch Hansen on Pascal Compilers*. Prentice-Hall, 1985.

[69] Kim Hazelwood and David Grove. Adaptive online context-sensitive inlining. In *Proceedings of the international symposium on Code generation and optimization*, pages 253–264, 2003.

[70] John Hennessy. Stanford integer benchmarks, 1988.

[71] Michael Hind and Anthony Pioli. Which pointer analysis should i use? In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 113–123. ACM Press, 2000.

[72] W. Holst and D. Szafron. Incremental table-based method dispatch for reflexive object-oriented languages, 1997.

[73] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *ECOOP '91, European Conference On Object-Oriented Programming*, 1991.

[74] Urs Hölzle, Craig Chambers, and David Ungar. Debugging optimized code with dynamic deoptimization. In *Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, pages 32–43. ACM Press, 1992.

[75] Urs Hölzle and David Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *Proceedings of the ACM SIGPLAN '94 conference on Programming language design and implementation*, pages 326–336. ACM Press, 1994.

[76] Kazuaki Ishizaki, Motohiro Kawahito, Toshiaki Yasue, Hideaki Komatsu, and Toshio Nakatani. A study of devirtualization techniques for a java just-in-time compiler. In *Proceedings of the conference on Object-oriented programming, systems, languages, and applications*, pages 294–310. ACM Press, 2000.

[77] Kazuaki Ishizaki, Motohiro Kawahito, Toshiaki Yasue, Mikio Takeuchi, Takeshi Ogasawara, Toshio Suganuma, Tamiya Onodera, Hideaki Komatsu, and Toshio Nakatani. Design, implementation, and evaluation of optimizations in a just-in-time compiler. In *Proceedings of the ACM 1999 conference on Java Grande*, pages 119–128. ACM Press, 1999.

[78] Jamie Jaworski. *Java 1.1 Developer's Guide*. Sams.net, second edition, 1997.

[79] Ralph E. Johnson, Justin O. Graver, and Laurance W. Zurawski. Ts: an optimizing compiler for smalltalk. In *Conference proceedings on Object-oriented programming systems, languages and applications*, pages 18–26. ACM Press, 1988.

[80] Guy L. Steele Jr. Growing a language, 1998.

[81] Kimberly Keeton, Russell Clapp, and Ashwini Nanda. Evaluating servers with commercial workloads. *IEEE Computer Society*, pages 29–32, 2003.

[82] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. Getting started with aspectj. *Communications of the ACM*, 44(10):59–65, October 2001.

[83] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.

[84] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[85] Chandra Krintz. Coupling on-line and off-line profile information to improve program performance.

[86] Doug Lea. Customization in c++. In *C++ Conference*, pages 301–314, 1990.

[87] Ondrej Lhotak and Laurie Hendren. Run-time evaluation of opportunities for object inlining in java. In *Proceedings of Java Grande / ISCOPE 2002*, 2002.

[88] Sheng Liang and Gilad Bracha. Dynamic class loading in the java virtual machine. In *Proceedings of the conference on Object-oriented programming, systems, languages, and applications*, pages 36–44. ACM Press, 1998.

[89] Chu-Cheow Lim and Andreas Stolcke. Sather language design and performance evaluation. Technical report, 1991.

[90] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification: Second Edition.* Addison Wesley, second edition, 1997.

[91] Yue Luo, Juan Rubio, Lizy Kurian John, Pattabi Seshadri, and Alex Mericas. Becnhmarking internet servers on superscalar machines. *IEEE Computer Society*, pages 34–40, 2003.

[92] Jeremy Manson and William PUgh. Semantics of multithreaded java, 2002.

[93] J. A. Mathew, P. D. Coddington, and K. A. Hawick. Analysis and Development of Java Grande Benchmarks. In *Proc. of the ACM 1999 Java Grande Conference, San Francisco.*, 1999.

[94] Steve McConnel. *Code Complete.* Microsoft Press, 1993.

[95] Wen mei W. Hwu, Scott A. Mahlke, William Y. Chen, Pohua P. Chang, Nancy J. Warter, Roger A. Bringmann, Roland G. Ouellette, Richard E. Hank, Tokuzo Kiyohara, Grant E. Haab, John G.Holm, and Daniel M. Lavery. The superblock: An effective technique for vliw and superscalar compilation. In *The Journal Of Supercomputing*, pages 229–248. Kluweer Academic Publishers, 1993.

[96] SUN Microsystems. The Java HotSpot Virtual Machine. http://java.sun.com/products/hotspot/whitepaper.html, 2001. Technical White Paper.

[97] Steven S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, 1998.

[98] Gilles Muller and Ulrik Pagh Schultz. Harissa: A hybrid approach to Java execution. *IEEE Software*, 16(2):44–51, 1999.

[99] Gail C. Murphy, Albert Lai, Robert J. Walker, and Martin P. Robillard. Separating features in source code: An exploratory study. In *International Conference on Software Engineering*, pages 275–284, 2001.

[100] Ted Neward. *Understanding Class.forName().* Javageeks.com, 2001.

[101] Patrick D. O'Brien, Daniel C. Halbert, and Michael F. Kilian. The trellis programming environment. In *OOPSLA '87: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 91–102, New York, NY, USA, 1987. ACM Press.

[102] Yale N. Patt and Sanjay J. Patel. *Introduction To Computing Systems: From Bits And Gates To C And Beyond.* McGraw-Hill, 2001.

[103] David A. Patterson and John L. Hennessy. *Computer Architecture A Quantitative Approach.* Morgan Kaufmann Publishers, INC, second edition, 1996.

[104] Karl Pettis and Robert C. Hansen. Profile guided code positioning. In *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, pages 16–27. ACM Press, 1990.

[105] P. Pominville, F. Qin, R. Vallee-Rai, L. Hendren, and C. Verbrugge. A framework for optimizing java using attributes. In *Proc. LNCS'2001*, pages 334–354, 2001.

[106] Todd A. Proebsting, Gregg Townsend, Patrick Bridges, John H. Hartman, Tim Newsham, and Scott A. Watterson. Toba: Java for applications — A Way Ahead of Time (WAT) compiler. pages 41–53, 1997.

[107] Barbara G. Ryder. Dimensions of precision in reference analysis of object-oriented programming languages. 2000.

[108] Nathanael Schrli, Stphane Ducasse, Oscar Nierstrasz, and Andrew P. Black. Traits: Composable units of behaviour.

[109] U. Schultz and C. Consel. Automatic program specialization for java, 2000.

[110] U. P. Schultz, J. L. Lawall, and C. Consel. Specialization patterns. In *Proceedings of the 15$^{th}$ IEEE International Conference on Automated Software Engineering (ASE 2000)*, pages 197–206, Grenoble, France, 2000. IEEE.

[111] Ulrik Pagh Schultz, Julia L. Lawall, Charles Consel, and Gilles Muller. Towards automatic specialization of Java programs. *Lecture Notes in Computer Science*, 1628:367–??, 1999.

[112] Timothy Sherwood, Suleyman Sair, and Brad Calder. Phase tracking and prediction.

[113] Yefim Shuf, Mauricio J. Serrano, Manish Gupta, and Jaswinder Pal Singh. Characterizing the memory behavior of java workloads: a structured view and opportunities for optimizations. In *SIGMETRICS/Performance*, pages 194–205, 2001.

[114] Kevin Skaron, Margaret Martonosi, David I. August, Mark D. Hill, David J. Lilja, and Vijay S. Pai. Challenges in computer architecture evaluation. *IEEE Computer*, pages 30–36, 2003.

[115] V. Sreedhar, M. Burke, and J. Choi. A framework for interprocedural optimization in the presence of dynamic class loading. In *Proc. PLDI'2000*, 2000.

[116] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley, 1997.

[117] Toshio Suganuma, Toshiaki Yasue, Motohiro Kawahito, Hideaki Komatsu, and Toshio Nakatani. A dynamic optimization framework for a java just-in-time compiler. In *Proceedings of the OOPSLA '01 conference on Object Oriented Programming Systems Languages and Applications*, pages 180–195. ACM Press, 2001.

[118] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. Practical virtual method call resolution for Java. *ACM SIGPLAN Notices*, 35(10):264–280, 2000.

[119] Antero Taivalsaari. Implementing a java virtual machine in the java programming language, 1998.

[120] Frank Tip and Peter F. Sweeney. Class hierarchy specialization. In *Proceedings of the 1997 ACM SIGPLAN conference on Object- oriented programming systems, languages and applications*, pages 271–285. ACM Press, 1997.

[121] Frank Tip and Peter F. Sweeney. Class hierarchy specialization. *Acta Informatica*, 36(12):927–982, 2000.

[122] Toshio Nakatani Toshio Suganuma, Toshiaki Yasue. An empirical study of method inlining for a java just-in-time compiler. In *Proceedings of the 2nd Java Virtual Machine Research and Technolgy Symposium*, August 2002.

[123] Raja Vallee-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework, 1999.

[124] Raja Vallee-Rai, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. Optimizing java bytecode using the soot framework: Is it feasible? In *Computational Complexity*, pages 18–34, 2000.

[125] Peter van der Linden. *Just Java 2*. Sun Microsystems Press, 1999.

[126] Eugen-Nicolae Volanschi, Charles Consel, Gilles Muller, and Crispin Cowan. Declarative specialization of object-oriented programs. In *Conference on Object-Oriented*, pages 286–300, 1997.

[127] Robert J. Walker, Elisa L. A. Baniassad, and Gail C. Murphy. An initial assessment of aspect-oriented programming. In *International Conference on Software Engineering*, pages 120–130, 1999.

[128] John Whaley. Dynamic optimization through the use of automatic runtime specialization. Master's thesis, Massachusetts Institute of Technology, May 1999.

[129] Peng Zhao and José Nelson Amaral. To inline or not to inline? enhanced inlining decisions. In *LCPC*, pages 405–419, 2003.

# Appendix A

# Raw Data for SPECjvm98 and SPECjbb2000 Benchmarks

This appendix contains all of the raw data collected from the experiments found in Section 9.3.2.

| Benchmark | Specializable Classes | Non-Specializable Classes | Total Classes (Loaded) | % Specializable Classes |
|---|---|---|---|---|
| 201_compress | 44 | 10 | 54 | 69% |
| 202_jess | 168 | 15 | 183 | 92% |
| 209_db | 38 | 10 | 48 | 79% |
| 213_javac | 132 | 41 | 173 | 76% |
| 222_mpegaudio | 75 | 9 | 84 | 89% |
| 227_mtrt | 54 | 13 | 67 | 81% |
| 228_jack | 78 | 10 | 88 | 89% |
| SPECjbb2000 | 135 | 15 | 150 | 90% |
| Total | 724 | 123 | 847 | 85% |

Table A.1: Static Number of Specializable Classes

| Benchmark | Specializable Methods | Non-Specializable Methods | Total Methods (Loaded) | % Specializable |
|---|---|---|---|---|
| 201_compress | 54 | 157 | 211 | 26% |
| 202_jess | 198 | 441 | 639 | 31% |
| 209_db | 49 | 179 | 228 | 21% |
| 213_javac | 215 | 686 | 901 | 24% |
| 222_mpegaudio | 92 | 279 | 371 | 25% |
| 227_mtrt | 69 | 265 | 334 | 21% |
| 228_jack | 105 | 339 | 444 | 24% |
| SPECjbb2000 | 180 | 718 | 898 | 20% |
| Total | 962 | 3064 | 4026 | 24% |

Table A.2: Static Number of Specializable Methods

| Benchmark | Devirtual-izable Call Sites | Non-Devirtualizable Call Sites | Total Call Sites | % Devirtualizable Call Sites |
|---|---|---|---|---|
| 201_compress | 62 | 2280 | 2342 | 2.6% |
| 202_jess | 206 | 5856 | 6062 | 3.4% |
| 209_db | 57 | 2475 | 2532 | 2.3% |
| 213_javac | 274 | 7867 | 8141 | 3.6% |
| 222_mpegaudio | 109 | 2949 | 3058 | 3.6% |
| 227_mtrt | 77 | 3750 | 3827 | 2.0% |
| 228_jack | 164 | 5194 | 5358 | 3.1% |
| SPECjbb2000 | 213 | 8077 | 8290 | 2.6% |
| Total | 1162 | 38448 | 39610 | 2.9% |

Table A.3: Static Number of Specializable Call Sites

| Benchmark | Inlinable Call Sites | Non-Inlinable Call Sites | Total Call Sites | Percent Inlinable Call Sites |
|---|---|---|---|---|
| 201_compress | 51 | 11 | 62 | 82% |
| 202_jess | 181 | 25 | 206 | 88% |
| 209_db | 46 | 11 | 57 | 81% |
| 213_javac | 137 | 137 | 274 | 50% |
| 222_mpegaudio | 78 | 31 | 109 | 72% |
| 227_mtrt | 63 | 14 | 77 | 82% |
| 228_jack | 78 | 86 | 164 | 48% |
| SPECjbb2000 | 177 | 36 | 213 | 83% |
| Total | 811 | 351 | 1162 | 70% |

Table A.4: Static Number of Inlinable Specializable Methods

# Appendix B

# Trademarks

Jikes, AIX, and PowerPC are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both.

Java and all Java-based trademarks are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.